KANRO TOMOYA

2024 Edition

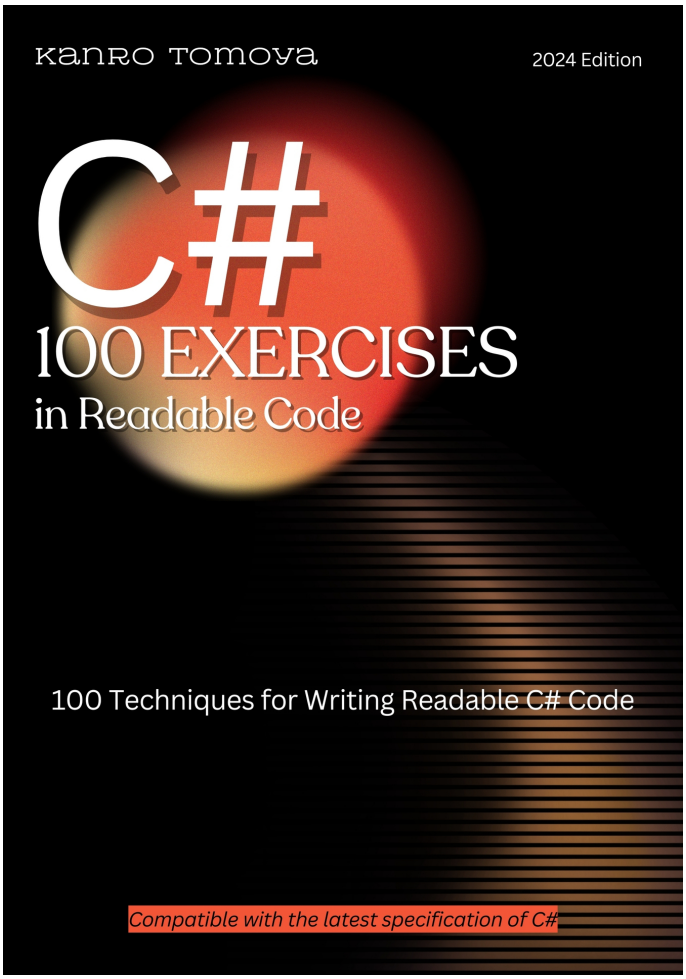
C#

100 EXERCISES

in Readable Code

100 Techniques for Writing Readable C# Code

Compatible with the latest specification of C#


KANRO TOMOYA
2024 Edition
C#
100 EXERCISES
in Readable Code
100 Techniques for Writing Readable C# Code
Compatible with the latest specification of C#

Index

Use var for local variable declarations when the type is obvious.
Utilize using statements to manage resource cleanup.
Leverage async and await for asynchronous programming.
Use LINQ for querying collections in a readable manner.
Implement properties instead of public fields for encapsulation.
Name variables with meaningful and descriptive names.
Use PascalCase for class names and method names
Use camelCase for local variables and method parameters
Prefix boolean variables with "is" or "has" to indicate their type.
Avoid abbreviations in names unless they are widely understood.
Avoid using single-letter variable names except in loops.
Use specific names instead of generic ones like data or info.
Avoid using magic numbers; use named constants instead
Ensure method names clearly describe their functionality
Use namespaces to avoid name collisions and organize code.
Use StringBuilder for concatenating large strings.
Utilize null-coalescing operator ?? to provide default values.
Use nameof operator for argument validation.
Leverage pattern matching for cleaner type checks
Use expression-bodied members for simple methods and properties
Comment on complex logic to explain the reasoning.
Document the purpose of public methods and classes.
Use TODO comments to indicate areas for future improvement.
Comment on any workarounds or hacks in the code.
Explain the Intent Behind Non-Obvious Code
Keep Comments Up-to-Date with Code Changes
Avoid redundant comments that state the obvious.
Use XML documentation comments for public APIs.
Write comments in complete sentences for clarity.

Use comments to explain why, not what.
Use readonly for fields that should not be modified after initialization
Leverage auto-properties for simple property declarations
Use tuples for returning multiple values from a method.
Utilize extension methods to add functionality to existing types.
Use delegates and events for implementing the observer pattern.
Keep lines of code within a reasonable length.
Use consistent indentation and formatting.
Group related code together logically.
Use whitespace to separate logical sections of code.
Align similar code vertically for better readability.
Use if-else statements instead of nested ternary operators.
Avoid deep nesting by using guard clauses.
Use switch statements for multiple conditions.
Break long methods into smaller, more manageable ones.
Use foreach instead of for when iterating over collections.
Use try-catch blocks to handle exceptions gracefully.
Avoid empty catch blocks; at least log the exception.
Use finally blocks to clean up resources.
Throw specific exceptions instead of generic ones.
Use using statements to ensure proper disposal of resources.
Use const for values that never change.
Declare variables as close to their usage as possible.
Use meaningful names for all variables
Avoid using the same variable for multiple purposes
Use var when the type is clear from the context.
Limit the scope of variables to the smallest possible.
Use descriptive names for loop counters.
Avoid global variables; use class fields or properties instead.
Use readonly for variables that should not change after initialization.

Initialize variables where they are declared.
Ensure each method performs a single, well-defined task.
Avoid long methods; break them into smaller ones.
Use helper methods to encapsulate complex logic.
Refactor code to eliminate duplication.
Use meaningful method names that describe their purpose.
Use async and await for asynchronous operations.
Leverage LINQ for querying and manipulating collections.
Use yield return for implementing custom iterators.
Utilize Task and Task<T> for parallel programming.
Use lock statements to handle concurrency issues.
Break down complex expressions into multiple lines.
Use intermediate variables to store results of sub-expressions.
Use parentheses to make the order of operations clear.
Avoid chaining multiple method calls in a single statement.
Use descriptive variable names for intermediate results.
Extract unrelated logic into separate methods.
Use helper methods to encapsulate common functionality.
Refactor large methods to improve readability.
Use design patterns to solve common problems.
Encapsulate complex logic within well-named methods.
Use async and await for asynchronous programming.
Leverage LINQ for querying collections.
Use using statements for resource management.
Utilize pattern matching for cleaner code.
Use expression-bodied members for simple methods.
Choose the appropriate collection type for your data.
Use Dictionary for key-value pairs.
Use List for ordered collections.
Use HashSet for unique elements.

Use Queue and Stack for FIFO and LIFO collections.
Use try-catch blocks to handle exceptions.
Log exceptions for debugging purposes.
Use custom exceptions for specific error conditions.
Validate method arguments to prevent errors.
Use finally blocks to clean up resources.
Write reusable methods for common tasks.
Use interfaces to define contracts for classes.
Leverage inheritance to reuse code.
Use generics to create flexible and reusable code.
Encapsulate common functionality in utility classes.
Use dependency injection to manage dependencies.
Write unit tests to ensure code correctness.

Introduction

◆

Welcome to this guide, which aims to enhance your C# programming skills by providing 100 practical techniques for writing cleaner, more readable code.

This book focuses on utilizing C#-specific features and best practices to ensure that your code is as understandable as possible for others.

Each technique is designed to help you write code that is not only functional but also clear and maintainable.

Included are examples of both good and bad coding practices, structured in a way that makes it easy to grasp the principles being discussed.

Through these examples, you'll learn how to transform complex and confusing code into something that is simple and elegant.

Dive in and start improving your coding habits today!

1

Use var for local variable declarations when the type is obvious.

Simplify code readability by using 'var' for local variable declarations when the type is clear from the context.

Using 'var' in local variable declarations helps make the code cleaner and easier to read, especially when the variable type is evident from the right-hand side of the assignment.

< Good Code >

```
var customerList = new List<Customer>(); // The type List<Customer> is obvious from the right-hand side.
var totalAmount = CalculateTotal(orderItems); // The return type of CalculateTotal is clear from its method definition.
```

<Bad Code>

```
List<Customer> customerList = new List<Customer>(); // Redundant type specification.
decimal totalAmount = CalculateTotal(orderItems); // The type is obvious from the method signature.
```

In the good example, using 'var' makes the code shorter and easier to read without sacrificing clarity. The type of the variable is still clear from the context, which improves maintainability. In the bad example, the explicit type declaration adds unnecessary verbosity, making the code harder to read without adding value.

<Memo>
C# 3.0 introduced 'var' as part of the language's type inference capabilities, which allows the compiler to deduce the type of a variable from its initializer.

2

Utilize using statements to manage resource cleanup.

Ensure proper resource management and cleanup by using 'using' statements to handle IDisposable objects.

The 'using' statement in C# ensures that IDisposable objects are properly disposed of once they are no longer needed, which helps prevent resource leaks and improves code reliability.

< Good Code >

```
using (var connection = new SqlConnection(connectionString))
{
    connection.Open();
    // Use the connection
} // connection is automatically disposed here
```

<Bad Code>

```
var connection = new SqlConnection(connectionString);
connection.Open();
// Use the connection
connection.Dispose(); // Manual disposal, error-prone if exceptions occur before this line
```

In the good example, the 'using' statement ensures that the SqlConnection object is disposed of as soon as the block of code is exited, even if an exception occurs. This guarantees that resources are properly freed. In the

bad example, manual disposal is prone to errors and can lead to resource leaks if an exception prevents the Dispose method from being called.

<Memo>
The 'using' statement was introduced in C# 1.0 to simplify resource management by ensuring deterministic disposal of unmanaged resources.

3

Leverage async and await for asynchronous programming.

Using async and await keywords in C# to write asynchronous code that is easy to read and maintain.

The async and await keywords in C# allow you to write asynchronous code that looks similar to synchronous code, making it easier to read and understand.

< Good Code >

```
csharppublic async Task<string> FetchDataAsync()
{
    // Asynchronously fetch data from a web service
    using (HttpClient client = new HttpClient())
    {
        string result = await
client.GetStringAsync("https://example.com/data");
        return result;
    }
}
```

<Bad Code>

```
csharppublic Task<string> FetchDataAsync()
{
    // Asynchronously fetch data from a web service
    using (HttpClient client = new HttpClient())
    {
```

```
        Task<string> task =
client.GetStringAsync("https://example.com/data");
        task.Wait(); // Blocking call, defeats the purpose of async
        return task;
    }
}
```

In the good example, the await keyword is used to asynchronously wait for the GetStringAsync method to complete, without blocking the thread. This makes the code more readable and efficient. In the bad example, the Wait method is used, which blocks the thread and negates the benefits of asynchronous programming.

<Memo>
The async keyword was introduced in C# 5.0, along with the await keyword, to simplify asynchronous programming and improve code readability.

4

Use LINQ for querying collections in a readable manner.

Utilize LINQ (Language Integrated Query) to perform queries on collections in a concise and readable way.

LINQ provides a powerful syntax for querying collections, making the code more readable and expressive compared to traditional loops and conditionals.

< Good Code >

```
csharpvar numbers = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
// Use LINQ to find even numbers
var evenNumbers = numbers.Where(n => n % 2 == 0).ToList();
```

<Bad Code>

```
csharpvar numbers = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
var evenNumbers = new List<int>();
// Use a loop to find even numbers
foreach (var number in numbers)
{
    if (number % 2 == 0)
    {
        evenNumbers.Add(number);
    }
}
```

In the good example, LINQ is used to filter the list of numbers, making the code concise and easy to understand. The Where method is used to specify the condition for filtering. In the bad example, a loop and conditional statements are used, which makes the code longer and harder to read.

<Memo>
LINQ was introduced in C# 3.0 and provides a consistent model for working with data across various sources, such as collections, databases, and XML.

5

Implement properties instead of public fields for encapsulation.

Use properties to encapsulate fields and provide controlled access.

Encapsulation is a fundamental principle in object-oriented programming. Using properties instead of public fields helps to protect the internal state of an object and allows for validation, logging, and other logic to be added when getting or setting values.

< Good Code >

```
csharppublic class Person
{
   // Private field
   private string name;
   // Public property with encapsulation
   public string Name
   {
      get { return name; }
      set
      {
         if (!string.IsNullOrEmpty(value))
         {
            name = value;
         }
      }
   }
}
```

<Bad Code>

```
csharppublic class Person
{
    // Public field without encapsulation
    public string name;
}
```

In the good example, the Name property encapsulates the name field, allowing for validation logic to be added when setting the value. This ensures that the name cannot be set to a null or empty string. In the bad example, the name field is public, meaning it can be directly accessed and modified without any control, potentially leading to invalid states.

<Memo>
Properties in C# can also be auto-implemented, which simplifies the syntax when no additional logic is needed in the getter or setter.

6

Name variables with meaningful and descriptive names.

Choose variable names that clearly describe their purpose and use.

Meaningful and descriptive variable names make code more readable and maintainable. They help other developers understand the purpose of the variable without needing additional comments or documentation.

< Good Code >

```
csharppublic class OrderProcessor
{
    public void ProcessOrder(int orderId)
    {
        // Meaningful variable name
        int numberOfItems = GetNumberOfItems(orderId);
        // Process the order
    }
    private int GetNumberOfItems(int orderId)
    {
        // Simulate getting the number of items for the order
        return 5;
    }
}
```

<Bad Code>

```
csharppublic class OrderProcessor
{
    public void ProcessOrder(int id)
```

```
    {
        // Non-descriptive variable name
        int n = GetNumberOfItems(id);
        // Process the order
    }
    private int GetNumberOfItems(int id)
    {
        // Simulate getting the number of items for the order
        return 5;
    }
}
```

In the good example, the variable numberOfItems clearly indicates what it represents, making the code easier to understand. In the bad example, the variable n is not descriptive, which can confuse other developers who read the code, as they would need to infer its purpose from the context.

<Memo>
Using meaningful variable names is part of writing self-documenting code, which reduces the need for extensive comments and makes the codebase easier to navigate.

7

Use PascalCase for class names and method names

Use PascalCase naming convention for classes and methods in C#. This improves code readability and follows the standard C# coding style.

When defining a class or method in C#, use PascalCase naming convention where the first letter of each word is capitalized.

< Good Code >

```
csharppublic class SampleClass
{
    public void DoSomething()
    {
        // Method body
    }
}
```

<Bad Code>

```
csharppublic class sampleClass
{
    public void dosomething()
    {
        // Method body
    }
}
```

PascalCase improves code readability by making it easier to distinguish between classes, methods, and other code elements. It follows the standard C# coding style and is widely adopted by the C# community.

<Memo>
PascalCase is also known as UpperCamelCase.

8

Use camelCase for local variables and method parameters

Use camelCase naming convention for local variables and method parameters in C#. This improves code readability and follows the standard C# coding style.

When defining local variables or method parameters in C#, use camelCase naming convention where the first letter of the first word is lowercase, and the first letter of each subsequent word is capitalized.

< Good Code >

```
csharppublic void DoSomething(int someParameter)
{
   int localVariable = 0;
   // Method body
}
```

<Bad Code>

```
csharppublic void DoSomething(int SomeParameter)
{
   int LocalVariable = 0;
   // Method body
}
```

camelCase improves code readability by making it easier to distinguish between local variables, method parameters, and other code elements. It

follows the standard C# coding style and is widely adopted by the C# community.

<Memo>
camelCase is also known as lowerCamelCase.

9

Prefix boolean variables with "is" or "has" to indicate their type.

Use "is" or "has" as prefixes for boolean variables to make their purpose clear.

Boolean variables should be easily identifiable by their names. Using prefixes like "is" or "has" helps indicate that the variable holds a true/false value.

< Good Code >

```
csharpbool isUserLoggedIn = true; // Indicates if the user is logged in
bool hasAccessRights = false; // Indicates if the user has access rights
```

<Bad Code>

```
csharpbool userLoggedIn = true; // Not immediately clear that this is a boolean
bool accessRights = false; // Not immediately clear that this is a boolean
```

In the good code example, the prefixes "is" and "has" clearly indicate that the variables are boolean, making the code more readable and understandable. In the bad code example, the lack of these prefixes makes it less obvious that the variables are boolean, potentially causing confusion.

<Memo>
The practice of prefixing boolean variables with "is" or "has" is common in many programming languages, not just C#. It helps maintain consistency and readability across different codebases.

10

Avoid abbreviations in names unless they are widely understood.

Use full words for variable and method names unless the abbreviation is universally recognized.

Abbreviations can make code harder to read and understand. Using full words ensures clarity, except when the abbreviation is well-known and widely accepted.

< Good Code >

```
csharpint userAge = 30; // Clear and understandable
string customerAddress = "123 Main St"; // Clear and understandable
```

<Bad Code>

```
csharpint usrAg = 30; // Abbreviation makes it unclear
string custAddr = "123 Main St"; // Abbreviation makes it unclear
```

In the good code example, using full words like "userAge" and "customerAddress" makes the code self-explanatory. In the bad code example, abbreviations like "usrAg" and "custAddr" can confuse readers who may not immediately understand what the variables represent.

<Memo>
Some widely accepted abbreviations, like "ID" for "identifier" or "URL" for "Uniform Resource Locator," are exceptions to this rule because they are universally understood in the programming community.

11

Avoid using single-letter variable names except in loops.

Use meaningful variable names to enhance code readability.

In C#, using single-letter variable names outside of loops can make code hard to understand. Instead, use descriptive names that convey the purpose of the variable.

< Good Code >

```
// Calculate the area of a rectangle
double length = 10.0;
double width = 5.0;
double area = length * width;
Console.WriteLine($"Area: {area}");
```

<Bad Code>

```
// Calculate the area of a rectangle
double l = 10.0;
double w = 5.0;
double a = l * w;
Console.WriteLine($"Area: {a}");
```

In the good example, length, width, and area clearly describe their purpose, making the code easier to read and understand. In the bad example, l, w, and a are ambiguous and require more effort to interpret.

<Memo>

Single-letter variable names are traditionally used in mathematics and are often seen in loop counters (e.g., i, j, k). However, for clarity, avoid their use elsewhere in your code.

12

Use specific names instead of generic ones like data or info.

Choose specific, descriptive names for variables to make code more understandable.

Generic variable names like data or info provide little context about what the variable represents. Using specific names helps others quickly grasp the variable's role and content.

< Good Code >

```
// Store user profile information
string userName = "JohnDoe";
int userAge = 30;
string userEmail = "john.doe@example.com";
```

<Bad Code>

```
// Store user profile information
string data1 = "JohnDoe";
int data2 = 30;
string data3 = "john.doe@example.com";
```

In the good example, userName, userAge, and userEmail explicitly describe what each variable holds, improving readability. In the bad example, data1, data2, and data3 are vague and make it hard to determine the purpose of each variable without additional context.

<Memo>

Using specific names also helps with code maintenance and debugging, as it reduces the likelihood of confusion and errors, making the development process more efficient.

13

Avoid using magic numbers; use named constants instead

Instead of using literal values (magic numbers) in your code, define named constants to improve code readability and maintainability.

When working with numeric values in code, it's often better to use named constants instead of literal values (magic numbers). This makes the code more self-documenting and easier to understand and maintain.

< Good Code >

```
csharppublic const double PI = 3.14159265358979;
public double CalculateCircumference(double radius)
{
    return 2 * PI * radius;
}
```

<Bad Code>

```
csharppublic double CalculateCircumference(double radius)
{
    return 2 * 3.14159265358979 * radius; // Magic number!
}
```

In the good code example, we define a constant PI with a descriptive name, making it clear what value is being used and why. In the bad code example, the literal value 3.14159265358979 is used directly, which can be confusing and harder to understand, especially if the same value is used in multiple places throughout the codebase.

Using named constants not only improves code readability but also makes it easier to change the value in the future if needed, as you only need to update the constant definition instead of searching for and replacing all occurrences of the literal value.

<Memo>
The term "magic number" refers to the use of literal values in code without any explanation or context, making it difficult for others (or even yourself in the future) to understand their meaning or purpose.

14

Ensure method names clearly describe their functionality

Choose descriptive and self-documenting method names that accurately convey the purpose and behavior of the method.

Method names should be clear and descriptive, accurately reflecting the functionality they provide. Well-named methods improve code readability and make it easier for other developers (or your future self) to understand the code's intent and purpose.

< Good Code >

```
csharppublic bool IsValidEmail(string email)
{
    // Email validation logic...
}
public string FormatDateString(DateTime date, string format)
{
    // Date formatting logic...
}
```

<Bad Code>

```
csharppublic bool Check(string input)
{
    // Email validation logic...
}
public string Transform(DateTime date, string pattern)
{
    // Date formatting logic...
```

```
}
```

In the good code example, the method names IsValidEmail and FormatDateString clearly convey their respective purposes: validating an email string and formatting a date string. In the bad code example, the method names Check and Transform are vague and do not provide enough context about what they do, making it harder to understand their functionality without digging into the implementation details.
When naming methods, aim for names that are concise yet descriptive, using verb-noun pairs or phrases that accurately describe the method's behavior. Avoid ambiguous or overly generic names that could be misinterpreted or require additional context to understand.

<Memo>
The principle of using clear and descriptive names is often referred to as the "Self-Documenting Code" principle, which emphasizes writing code that is easy to understand and maintain without relying heavily on external documentation or comments.

15

Use namespaces to avoid name collisions and organize code.

Namespaces help prevent name collisions and keep your code organized.

Using namespaces is a fundamental practice in C# to manage code by grouping related classes, interfaces, and functions together. It helps avoid name conflicts, especially in larger projects or when using external libraries.

< Good Code >

```
// Good example of using namespaces
namespace ProjectA.Utilities
{
    public class Logger
    {
        public void Log(string message)
        {
            Console.WriteLine(message);
        }
    }
}
namespace ProjectA.Services
{
    public class UserService
    {
        private Utilities.Logger _logger = new Utilities.Logger();
        public void CreateUser(string username)
        {
            _logger.Log($"User {username} created.");
        }
    }
```

```
}
```

<Bad Code>

```
// Bad example without namespaces
public class Logger
{
    public void Log(string message)
    {
        Console.WriteLine(message);
    }
}
public class UserService
{
    private Logger _logger = new Logger();
    public void CreateUser(string username)
    {
        _logger.Log($"User {username} created.");
    }
}
```

In the good example, namespaces ProjectA.Utilities and ProjectA.Services are used to group related classes together, preventing potential name collisions and improving code organization. In the bad example, the lack of namespaces can lead to conflicts and makes the code harder to manage in larger projects.

<Memo>

The concept of namespaces is not unique to C#. Many other programming languages like Java, C++, and Python also use namespaces or similar constructs to organize code and avoid name conflicts.

16

Use StringBuilder for concatenating large strings.

StringBuilder is more efficient than using string concatenation for handling large or numerous string operations.

String concatenation using the + operator can be inefficient in C# due to the immutable nature of strings. StringBuilder provides a more performant way to build large strings by minimizing memory overhead and execution time.

< Good Code >

```
// Good example using StringBuilder
using System.Text;
public class Example
{
    public string BuildLargeString()
    {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < 10000; i++)
        {
            sb.Append("This is line ");
            sb.Append(i);
            sb.Append("\n");
        }
        return sb.ToString();
    }
}
```

<Bad Code>

```
// Bad example using string concatenation
public class Example
{
    public string BuildLargeString()
    {
        string result = "";
        for (int i = 0; i < 10000; i++)
        {
            result += "This is line " + i + "\n";
        }
        return result;
    }
}
```

In the good example, StringBuilder is used to efficiently concatenate strings within a loop. This approach minimizes the creation of intermediate string objects, reducing memory usage and improving performance. The bad example uses simple string concatenation, which can lead to significant performance issues due to the immutable nature of strings, resulting in numerous intermediate objects and increased memory consumption.

<Memo>
StringBuilder is part of the System.Text namespace in .NET. It is specifically designed for scenarios where frequent modifications to string contents are required, making it a crucial tool for performance optimization in string manipulation tasks.

17

Utilize null-coalescing operator ?? to provide default values.

Use the null-coalescing operator (??) to assign default values to variables that might be null.

The null-coalescing operator (??) simplifies handling of null values by providing an easy way to specify a default value. This makes the code more readable and reduces the need for verbose null checks.

< Good Code >

```
string input = null;
string result = input ?? "Default Value";  // If input is null, result is assigned "Default Value"
```

<Bad Code>

```
string input = null;
string result;
if (input != null)
{
    result = input;
}
else
{
    result = "Default Value";
}  // Verbose and harder to read
```

The good example uses the null-coalescing operator to concisely handle the null case, making the code easier to read and understand. The bad example uses a traditional if-else structure, which is more verbose and harder to maintain.

<Memo>
The null-coalescing operator (??) was introduced in C# 2.0 and has since been a valuable tool for developers to manage null values efficiently.

18

Use nameof operator for argument validation.

Use the nameof operator to improve argument validation and exception handling, making the code clearer and less error-prone.

The nameof operator returns the name of a variable, type, or member as a string. It is particularly useful for argument validation in methods, enhancing code readability and reducing maintenance effort when refactoring.

< Good Code >

```
public void SetName(string name)
{
    if (string.IsNullOrWhiteSpace(name))
    {
        throw new ArgumentException("Parameter cannot be null or empty",
nameof(name));
    }
    // Further processing
}
```

<Bad Code>

```
public void SetName(string name)
{
    if (string.IsNullOrWhiteSpace(name))
    {
        throw new ArgumentException("Parameter cannot be null or empty",
"name");
```

```
    }
    // Further processing
}
```

The good example uses the nameof operator, which helps avoid hardcoding parameter names and reduces the risk of errors during refactoring. The bad example hardcodes the parameter name, making it less maintainable and prone to mistakes if the parameter name changes.

<Memo>
The nameof operator was introduced in C# 6.0, providing a more reliable way to refer to variable and parameter names in exception handling and logging.

19

Leverage pattern matching for cleaner type checks

Pattern matching simplifies complex type checks and improves code readability.

Pattern matching allows you to check the type of an object and perform different actions based on the type, all in a single expression.

< Good Code >

```
csharppublic static decimal CalculateArea(object shape)
{
    return shape switch
    {
        Circle c => Math.PI * c.Radius * c.Radius,
        Rectangle r => r.Length * r.Width,
        Square s => s.Side * s.Side,
        _ => throw new ArgumentException("Invalid shape", nameof(shape))
    };
}
```

<Bad Code>

```
csharppublic static decimal CalculateArea(object shape)
{
    if (shape is Circle c)
    {
        return Math.PI * c.Radius * c.Radius;
    }
    else if (shape is Rectangle r)
```

```
    {
        return r.Length * r.Width;
    }
    else if (shape is Square s)
    {
        return s.Side * s.Side;
    }
    else
    {
        throw new ArgumentException("Invalid shape", nameof(shape));
    }
}
```

The good code example uses pattern matching to check the type of the shape object and perform the appropriate area calculation in a single expression. This approach is more concise and easier to read than the traditional if-else statements used in the bad code example.

<Memo>
Pattern matching was introduced in C# 7.0 and has been further enhanced in subsequent versions.

20

Use expression-bodied members for simple methods and properties

---

Expression-bodied members provide a concise syntax for simple methods and properties.

Expression-bodied members allow you to define simple methods and properties using a lambda-like syntax, making the code more compact and readable.

< Good Code >

```
csharppublic class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public string GetGreeting() => $"Hello, {Name}!";
    public bool IsAdult() => Age >= 18;
}
```

<Bad Code>

```
csharppublic class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public string GetGreeting()
    {
        return $"Hello, {Name}!";
    }
    public bool IsAdult()
```

```
    {
        return Age >= 18;
    }
}
```

The good code example uses expression-bodied members for the GetGreeting method and the IsAdult method, making the code more concise and readable. The bad code example uses the traditional method syntax, which can be more verbose for simple methods.

<Memo>
Expression-bodied members were introduced in C# 6.0 and are particularly useful for implementing simple properties and methods.

21

Comment on complex logic to explain the reasoning.

Use comments to clarify complex logic in your code.

When writing complex logic, it is essential to add comments that explain the reasoning behind the code. This helps other developers understand the purpose and functionality of the code more easily.

< Good Code >

```
csharp// Calculate the factorial of a number using recursion
public int Factorial(int n)
{
    // Base case: if n is 0, return 1
    if (n == 0)
    {
        return 1;
    }
    // Recursive case: n * factorial of (n-1)
    return n * Factorial(n - 1);
}
```

<Bad Code>

```
csharppublic int Factorial(int n)
{
    if (n == 0)
    {
        return 1;
    }
```

```
    return n * Factorial(n - 1);
}
```

In the good code example, comments are used to explain the base case and the recursive case, making it clear why the code is written in this way. In the bad code example, the lack of comments makes it harder for others to understand the logic, especially if they are not familiar with recursion.

<Memo>
Comments should not state the obvious but should provide insight into the reasoning behind complex or non-intuitive code.

22

Document the purpose of public methods and classes.

Always document the purpose of public methods and classes.

Public methods and classes should have clear documentation that describes their purpose and usage. This helps other developers understand how to use them correctly and what to expect from them.

< Good Code >

```
csharp/// <summary>
/// Represents a customer in the system.
/// </summary>
public class Customer
{
    /// <summary>
    /// Gets or sets the customer's name.
    /// </summary>
    public string Name { get; set; }
    /// <summary>
    /// Calculates the discount for the customer based on their purchase history.
    /// </summary>
    /// <returns>The discount percentage.</returns>
    public double CalculateDiscount()
    {
        // Logic to calculate discount
        return 0.0;
    }
}
```

<Bad Code>

```
csharppublic class Customer
{
    public string Name { get; set; }
    public double CalculateDiscount()
    {
        return 0.0;
    }
}
```

In the good code example, XML documentation comments are used to describe the purpose of the Customer class and its CalculateDiscount method. This makes it clear to other developers what the class and method are for. In the bad code example, the lack of documentation means that other developers have to read and understand the code to figure out its purpose.

<Memo>
XML documentation comments can be used to generate API documentation automatically, making it easier to maintain and share documentation with other developers.

23

Use TODO comments to indicate areas for future improvement.

Use TODO comments to mark sections of code that need further work or enhancements.

TODO comments help developers identify parts of the code that require additional attention or improvements in the future.

< Good Code >

```
csharp// Calculate the sum of an array
public int SumArray(int[] numbers)
{
    int sum = 0;
    foreach (int number in numbers)
    {
        sum += number;
    }
    // TODO: Optimize this method for large arrays
    return sum;
}
```

<Bad Code>

```
csharppublic int SumArray(int[] numbers)
{
    int sum = 0;
    foreach (int number in numbers)
    {
        sum += number;
```

```
    }
    // This method might be slow for large arrays
    return sum;
}
```

In the good example, the TODO comment clearly indicates that the method should be optimized in the future. This helps other developers quickly understand that there is a known area for improvement. In the bad example, the comment is vague and does not provide a clear action item.

<Memo>
TODO comments can be easily searched and tracked using IDEs, making it simpler to manage and address them over time.

24

Comment on any workarounds or hacks in the code.

Always comment on any workarounds or hacks to explain why they are necessary.

Comments on workarounds or hacks provide context and reasoning, helping other developers understand the necessity and implications of such code.

< Good Code >

```
csharppublic string GetUserName(int userId)
{
    // Hack: Using a hardcoded user ID for demo purposes
    if (userId == 0)
    {
        return "DemoUser";
    }
    // TODO: Replace with actual database call
    return FetchUserNameFromDatabase(userId);
}
```

<Bad Code>

```
csharppublic string GetUserName(int userId)
{
    if (userId == 0)
    {
        return "DemoUser";
    }
    return FetchUserNameFromDatabase(userId);
```

```
}
```

In the good example, the comment explains that the hardcoded user ID is a temporary hack for demonstration purposes and indicates that it should be replaced with a proper database call. This provides clarity and context for future developers. In the bad example, the lack of comments leaves other developers guessing about the purpose and necessity of the hardcoded value.

<Memo>
Documenting workarounds and hacks can prevent potential bugs and misunderstandings, ensuring that temporary solutions are revisited and properly addressed in the future.

25

Explain the Intent Behind Non-Obvious Code

Clearly document the purpose and logic behind complex or non-intuitive code sections.

When working with intricate algorithms, data structures, or legacy code, it's essential to provide explanations for sections that may not be immediately understandable.

< Good Code >

```
csharp// Implement the Quicksort algorithm to sort the array in ascending order
private void QuickSort(int[] arr, int left, int right)
{
    int pivot;
    if (left < right)
    {
        pivot = Partition(arr, left, right);
        // Recursively sort the left and right partitions
        QuickSort(arr, left, pivot - 1);
        QuickSort(arr, pivot + 1, right);
    }
}
```

<Bad Code>

```
csharpprivate void ComplexMethod(int[] data)
{
    int a = 0, b = data.Length - 1;
```

```
    while (true)
    {
        while (a < b && data[a] < data[b])
            b--;
        if (a < b)
            Swap(data, a++, b);
        else
            break;
        while (a < b && data[a] < data[b])
            a++;
        if (a < b)
            Swap(data, a, b--);
        else
            break;
    }
}
```

The good code example clearly explains that the method implements the Quicksort algorithm for sorting an array in ascending order. The comments provide context for the recursive calls and their purpose. In contrast, the bad code example lacks any explanations, making it difficult to understand the algorithm's intent or the logic behind the code.

<Memo>
Commenting non-obvious code not only aids in understanding but also facilitates future maintenance and collaboration.

26

Keep Comments Up-to-Date with Code Changes

Ensure that comments accurately reflect the current state of the code by updating them whenever changes are made.

As code evolves through iterations and refactoring, comments can become outdated or misleading if not properly maintained. Keeping comments synchronized with the code is crucial for maintaining clarity and avoiding confusion.

< Good Code >

```
csharp// Calculate the area of a rectangle
public double CalculateArea(double length, double width)
{
    // Multiply length and width to get the area
    return length * width;
}
```

<Bad Code>

```
csharp// Calculate the volume of a sphere
public double CalculateVolume(double radius)
{
    // Use the formula: V = 4/3 * pi * r^3
    return (4.0 / 3.0) * Math.PI * Math.Pow(radius, 3);
}
```

In the good code example, the comments accurately describe the purpose and implementation of the CalculateArea method. However, in the bad code example, the comment incorrectly states that the method calculates the volume of a sphere, when it actually calculates the volume of a sphere using the correct formula. Keeping comments up-to-date with code changes is essential to avoid such discrepancies and maintain code clarity.

<Memo>
Outdated comments can be more harmful than no comments at all, as they can mislead developers and introduce bugs or misunderstandings.

27

Avoid redundant comments that state the obvious.

Write comments that add value and avoid stating what the code already clearly expresses.

Comments should provide insight into the code's purpose or complex logic, not restate what is evident from the code itself.

< Good Code >

```
csharp// Calculate the area of a circle
double radius = 5.0;
double area = Math.PI * Math.Pow(radius, 2); // Area = πr^2
```

<Bad Code>

```
csharp// Set radius to 5.0
double radius = 5.0;
// Calculate the area of the circle using the formula πr^2
double area = Math.PI * Math.Pow(radius, 2);
```

In the good example, the comment explains the purpose of the code (calculating the area of a circle) and provides a useful formula. In the bad example, the comments merely restate what the code is doing, which is already clear from the code itself.

<Memo>
Redundant comments can clutter the code and make it harder to maintain. Focus on writing self-explanatory code and use comments to explain the

"why" rather than the "what."

28

Use XML documentation comments for public APIs.

Utilize XML documentation comments to provide detailed information about public methods, properties, and classes.

XML documentation comments help generate comprehensive documentation and provide useful information to developers using your APIs.

< Good Code >

```
csharp/// <summary>
/// Calculates the area of a circle.
/// </summary>
/// <param name="radius">The radius of the circle.</param>
/// <returns>The area of the circle.</returns>
public double CalculateCircleArea(double radius)
{
    return Math.PI * Math.Pow(radius, 2);
}
```

<Bad Code>

```
csharppublic double CalculateCircleArea(double radius)
{
    return Math.PI * Math.Pow(radius, 2);
}
```

The good example uses XML documentation comments to describe the method's purpose, parameters, and return value, which aids in generating documentation and helps other developers understand how to use the method. The bad example lacks these comments, making it harder for others to understand the method's usage without looking at the implementation.

<Memo>
XML documentation comments can be processed by tools like Sandcastle or Doxygen to generate detailed API documentation, which is especially useful for large projects and libraries.

29

Write comments in complete sentences for clarity.

Using complete sentences in comments helps ensure clarity and readability for all developers.

When writing comments in your C# code, using complete sentences can provide clear and comprehensive explanations. This practice helps other developers understand the context and purpose of the code more easily.

< Good Code >

```
// This method calculates the factorial of a given number using recursion.
// It returns the factorial value as an integer.
public int CalculateFactorial(int number)
{
    if (number <= 1)
    {
        return 1; // Base case: factorial of 1 or 0 is 1.
    }
    else
    {
        return number * CalculateFactorial(number - 1); // Recursive call to
calculate factorial.
    }
}
```

<Bad Code>

```
// factorial calculation
public int CalculateFactorial(int number)
```

```
{
    if (number <= 1)
    {
        return 1; // base case
    }
    else
    {
        return number * CalculateFactorial(number - 1); // recursion
    }
}
```

In the good example, the comments provide complete sentences that clearly explain the purpose and logic of the code. In the bad example, the comments are brief and lack detail, making it harder to understand the code's intent.Complete sentences in comments ensure that the code's functionality and logic are communicated effectively, reducing the likelihood of misunderstandings and errors.

<Memo>
Using complete sentences in comments can also aid non-native English speakers by providing full context, making it easier to understand and translate the code explanations.

30

Use comments to explain why, not what.

Comments should focus on explaining the reasoning behind the code rather than describing what the code does.

In C# programming, it's important to use comments to explain the reasoning and thought process behind the code. This helps other developers understand the motivations for certain decisions, making maintenance and updates easier.

< Good Code >

```
// Using a dictionary to store user data for quick lookup by username.
// This approach ensures efficient retrieval compared to a list.
Dictionary<string, User> users = new Dictionary<string, User>();
// Adding a new user to the dictionary.
// The username must be unique, ensuring no duplicate entries.
users.Add("john_doe", new User("John", "Doe"));
```

<Bad Code>

```
// create dictionary
Dictionary<string, User> users = new Dictionary<string, User>();
// add user
users.Add("john_doe", new User("John", "Doe"));
```

In the good example, the comments explain why a dictionary is used and why the username must be unique. This provides context and rationale for the code's structure. In the bad example, the comments merely describe the actions being taken, which is redundant because the code itself is self-explanatory.By explaining the reasons behind the code, comments become

more informative and valuable, helping developers understand the design choices and making future modifications easier.

<Memo>
Explaining the "why" in comments can also document design patterns or specific algorithms used, providing insight into the overall architecture and making the codebase more robust and maintainable.

31

Use readonly for fields that should not be modified after initialization

---

Use the readonly keyword to prevent fields from being modified after initialization, improving code clarity and preventing accidental mutations.

Fields marked as readonly can only be assigned in the constructor or during initialization. This ensures their values remain constant throughout the object's lifetime.

< Good Code >

```
csharppublic class Circle
{
    private readonly double _radius;
    private readonly double _pi = 3.14159;
    public Circle(double radius)
    {
        _radius = radius;
    }
    public double CalculateArea()
    {
        return _pi * _radius * _radius;
    }
}
```

<Bad Code>

```
csharppublic class Circle
{
    private double _radius;
```

```
    private double _pi = 3.14159;
    public Circle(double radius)
    {
        _radius = radius;
    }
    public double CalculateArea()
    {
        // Accidentally modifying _pi would lead to incorrect results
        _pi = 3.0;
        return _pi * _radius * _radius;
    }
}
```

In the good code example, _radius and _pi are marked as readonly, ensuring their values cannot be modified after initialization. This prevents accidental mutations and makes the code more readable and maintainable. In the bad code example, _pi can be accidentally modified, leading to incorrect results.

<Memo>
The readonly keyword is a compile-time constraint, meaning the compiler enforces the rule. It does not prevent reflection or unsafe code from modifying the field.

32

Leverage auto-properties for simple property declarations

Use auto-properties for simple property declarations to reduce boilerplate code and improve code readability.

Auto-properties automatically generate the backing field and property accessors, reducing the amount of code needed for simple property declarations.

< Good Code >

```
csharppublic class Person
{
   public string Name { get; set; }
   public int Age { get; set; }
}
```

<Bad Code>

```
csharppublic class Person
{
   private string _name;
   private int _age;
   public string Name
   {
      get { return _name; }
      set { _name = value; }
   }
   public int Age
   {
```

```
        get { return _age; }
        set { _age = value; }
    }
}
```

In the good code example, auto-properties are used, reducing the amount of code needed for simple property declarations. The bad code example shows the traditional way of declaring properties, which requires more code and can be harder to read and maintain.

<Memo>
Auto-properties were introduced in C# 3.0 and have become a widely adopted practice for simple property declarations. They can also be combined with other features like data annotations or property initializers.

33

Use tuples for returning multiple values from a method.

A simple technique to enhance code readability by using tuples to return multiple values from a method in C#.

Using tuples allows methods to return more than one value without the need for creating custom classes or out parameters, making the code cleaner and easier to read.

< Good Code >

```
public (int Sum, int Product) Calculate(int a, int b)
{
    int sum = a + b;
    int product = a * b;
    return (sum, product); // Return values as a tuple
}
public void Example()
{
    var result = Calculate(3, 4);
    Console.WriteLine($"Sum: {result.Sum}, Product: {result.Product}"); //
Access tuple elements by name
}
```

<Bad Code>

```
public void Calculate(int a, int b, out int sum, out int product)
{
    sum = a + b;
    product = a * b;
```

```
}
public void Example()
{
    int sum, product;
    Calculate(3, 4, out sum, out product);
    Console.WriteLine($"Sum: {sum}, Product: {product}"); // Uses out
parameters
}
```

Using tuples makes the code more concise and easier to understand by avoiding out parameters, which can be confusing and error-prone. Tuples provide named elements, enhancing readability.

<Memo>
Tuples were introduced in C# 7.0 and provide a lightweight way to group multiple values. They are commonly used for short-lived data structures that do not require a separate class.

34

Utilize extension methods to add functionality to existing types.

Extension methods allow you to add new methods to existing types without modifying their source code or creating derived types.

Extension methods provide a powerful way to extend the functionality of existing types in C#, improving code readability and organization by enabling the addition of methods to types you do not own.

< Good Code >

```
public static class StringExtensions
{
    public static bool IsPalindrome(this string str)
    {
        int length = str.Length;
        for (int i = 0; i < length / 2; i++)
        {
            if (str[i] != str[length - i - 1])
                return false;
        }
        return true;
    }
}
public void Example()
{
    string word = "level";
    bool isPalindrome = word.IsPalindrome(); // Call the extension method
    Console.WriteLine($"Is '{word}' a palindrome? {isPalindrome}");
}
```

<Bad Code>

```
public static bool IsPalindrome(string str)
{
    int length = str.Length;
    for (int i = 0; i < length / 2; i++)
    {
        if (str[i] != str[length - i - 1])
            return false;
    }
    return true;
}
public void Example()
{
    string word = "level";
    bool isPalindrome = IsPalindrome(word); // No extension method
    Console.WriteLine($"Is '{word}' a palindrome? {isPalindrome}");
}
```

Extension methods improve the readability of code by enabling a more natural syntax, similar to instance method calls. This avoids modifying existing types or cluttering global namespaces with utility functions.

<Memo>

Extension methods were introduced in C# 3.0. They are static methods but can be called as if they were instance methods on the extended type, providing syntactic sugar for more intuitive code.

35

Use delegates and events for implementing the observer pattern.

Delegates and events in C# provide a robust way to implement the observer pattern, allowing objects to notify other objects about changes.

The observer pattern is a design pattern in which an object, known as the subject, maintains a list of its dependents, called observers, and notifies them of any state changes. In C#, delegates and events are used to implement this pattern efficiently.

< Good Code >

```
csharpusing System;
public class Subject
{
   // Declare the delegate (if using non-generic pattern).
     public delegate void Notify();
   // Declare the event using the delegate.
     public event Notify OnNotify;
   public void ChangeState()
   {
      Console.WriteLine("State has changed.");
      // Notify all observers about the state change.
         OnNotify?.Invoke();
   }
}
public class Observer
{
   public void Subscribe(Subject subject)
   {
      subject.OnNotify += Update;
```

```
    }
    private void Update()
    {
        Console.WriteLine("Observer has been notified of the state change.");
    }
}
public class Program
{
    public static void Main()
    {
        Subject subject = new Subject();
        Observer observer = new Observer();
        observer.Subscribe(subject);
        subject.ChangeState();
    }
}
```

<Bad Code>

```
csharpusing System;
public class Subject
{
    public Action OnNotify;  // Using Action instead of a proper delegate
    public void ChangeState()
    {
        Console.WriteLine("State has changed.");
        if (OnNotify != null)
        {
            OnNotify();  // No null-conditional operator
        }
    }
}
public class Observer
{
    public void Subscribe(Subject subject)
    {
        subject.OnNotify += Update;
```

```
    }
    private void Update()
    {
        Console.WriteLine("Observer has been notified of the state change.");
    }
}
public class Program
{
    public static void Main()
    {
        Subject subject = new Subject();
        Observer observer = new Observer();
        observer.Subscribe(subject);
        subject.ChangeState();
    }
}
```

In the good example, the delegate and event are explicitly declared, making the code more readable and maintainable. The null-conditional operator (?.) is used to safely invoke the event. In the bad example, an Action delegate is used, which is less descriptive, and the null check is done manually, which is more error-prone.

<Memo>
The observer pattern is widely used in implementing distributed event-handling systems, such as in the Model-View-Controller (MVC) architectural pattern.

36

Keep lines of code within a reasonable length.

Maintaining a reasonable line length in your code improves readability and helps prevent horizontal scrolling.

Long lines of code can be difficult to read and maintain. Keeping lines within a reasonable length (typically 80-100 characters) ensures that the code is more readable and easier to work with, especially in environments with limited screen space.

< Good Code >

```
csharppublic class Example
{
    public void ProcessData(string input)
    {
        if (string.IsNullOrEmpty(input))
        {
            throw new ArgumentException("Input cannot be null or empty",
nameof(input));
        }
        // Process the input data
        string processedData = input.Trim().ToUpper();
        Console.WriteLine($"Processed Data: {processedData}");
    }
}
```

<Bad Code>

```
csharppublic class Example
```

```
{
    public void ProcessData(string input) { if (string.IsNullOrEmpty(input))
{ throw new ArgumentException("Input cannot be null or empty",
nameof(input)); } string processedData = input.Trim().ToUpper();
Console.WriteLine($"Processed Data: {processedData}"); }
}
```

In the good example, the code is broken down into multiple lines, making it easier to read and understand. Each logical step is on its own line, which improves clarity. In the bad example, all the code is crammed into a single line, making it hard to read and understand.

<Memo>
The practice of keeping lines of code short dates back to the early days of programming when screens and printouts had limited width. This practice continues to be relevant for improving code readability and maintainability.

37

Use consistent indentation and formatting.

Consistent indentation and formatting make code easier to read and maintain.

Proper indentation and formatting help in understanding the structure and flow of the code. It is essential for collaboration and long-term maintenance.

< Good Code >

```
csharppublic class Example
{
    public void PrintMessage()
    {
        if (true)
        {
            Console.WriteLine("Hello, World!");
        }
    }
}
```

<Bad Code>

```
csharppublic class Example
{
public void PrintMessage()
{
if (true)
{
```

```
Console.WriteLine("Hello, World!");
}
}
}
```

In the good example, the code is indented consistently, making it clear where each block starts and ends. In the bad example, inconsistent indentation makes it difficult to follow the code structure.

<Memo>
Most modern IDEs and code editors have features to automatically format code according to a set of style guidelines, which can help maintain consistency.

38

Group related code together logically.

Grouping related code together improves readability and maintainability.

Logical grouping of code segments, such as related methods or properties, helps in understanding the functionality and flow of the program.

< Good Code >

```
csharppublic class Example
{
    private string message;
    public Example(string msg)
    {
        message = msg;
    }
    public void PrintMessage()
    {
        Console.WriteLine(message);
    }
    private void HelperMethod()
    {
        // Some helper functionality
    }
}
```

<Bad Code>

```
csharppublic class Example
{
    private void HelperMethod()
    {
```

```
        // Some helper functionality
    }
    public void PrintMessage()
    {
        Console.WriteLine(message);
    }
    private string message;
    public Example(string msg)
    {
        message = msg;
    }
}
```

In the good example, related code (fields, constructors, methods) is grouped together logically, making it easier to understand the class structure. In the bad example, the code is scattered, making it harder to follow the class's purpose and functionality.

<Memo>
Using regions in C# can help in organizing code by grouping related sections together, which can be collapsed or expanded in the IDE for better readability.

39

Use whitespace to separate logical sections of code.

Using whitespace effectively can make your code more readable by clearly separating different logical sections.

Whitespace helps in visually breaking down the code into manageable sections, making it easier to understand and maintain.

< Good Code >

```
csharppublic class Calculator
{
    // Adds two numbers
    public int Add(int a, int b)
    {
        return a + b;
    }
    // Subtracts the second number from the first
    public int Subtract(int a, int b)
    {
        return a - b;
    }
    // Multiplies two numbers
    public int Multiply(int a, int b)
    {
        return a * b;
    }
}
```

<Bad Code>

```
csharppublic class Calculator
{
    public int Add(int a, int b)
    {
        return a + b;
    }
    public int Subtract(int a, int b)
    {
        return a - b;
    }
    public int Multiply(int a, int b)
    {
        return a * b;
    }
}
```

In the good example, whitespace is used to separate each method, making it clear where one method ends and another begins. This improves readability and maintainability. In the bad example, the lack of whitespace makes the code look cluttered and harder to read.

<Memo>
Whitespace in code does not affect the execution of the program but significantly impacts readability and maintainability.

40

Align similar code vertically for better readability.

Aligning similar lines of code vertically can make patterns and structures in your code more apparent.

Vertical alignment helps in quickly identifying related lines of code, making it easier to spot errors and understand the code structure.

< Good Code >

```
csharppublic class Person
{
    public string FirstName { get; set; }
    public string LastName  { get; set; }
    public int    Age       { get; set; }
    public string Address   { get; set; }
}
```

<Bad Code>

```
csharppublic class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int Age { get; set; }
    public string Address { get; set; }
}
```

In the good example, the properties are aligned vertically, making it easy to scan and compare them. This alignment helps in quickly identifying the structure and any potential issues. In the bad example, the properties are not aligned, making the code harder to read and understand.

<Memo>
Vertical alignment is particularly useful in data structures and configuration files where similar elements are defined repeatedly.

41

Use if-else statements instead of nested ternary operators.

Simplify code readability by avoiding complex nested ternary operators and using if-else statements.

When writing C# code, using nested ternary operators can make the code difficult to read and understand. Using if-else statements improves clarity and maintainability.

< Good Code >

```
// Good Example: Using if-else statements
int value = 10;
string result;
if (value > 0)
{
    result = "Positive";
}
else if (value < 0)
{
    result = "Negative";
}
else
{
    result = "Zero";
}
Console.WriteLine(result); // Output: Positive
```

<Bad Code>

```
// Bad Example: Using nested ternary operators
int value = 10;
string result = value > 0 ? "Positive" : value < 0 ? "Negative" : "Zero";
Console.WriteLine(result); // Output: Positive
```

Using if-else statements, as shown in the good example, makes the logic clear and easy to follow. Each condition is explicitly separated, which helps in understanding the code flow. On the other hand, the nested ternary operator in the bad example, while concise, is harder to read and understand, especially for those not familiar with the syntax or for more complex conditions.

<Memo>
The ternary operator is also known as the conditional operator and is the only operator in C# that takes three operands. While it can be useful for simple conditions, it should be used sparingly to maintain code readability.

42

Avoid deep nesting by using guard clauses.

Improve code readability and reduce complexity by using guard clauses to handle edge cases early.

Deep nesting in C# code can make it difficult to read and maintain. By using guard clauses, you can simplify the structure and enhance the clarity of your code.

< Good Code >

```
// Good Example: Using guard clauses
void ProcessOrder(Order order)
{
    if (order == null)
    {
        throw new ArgumentNullException(nameof(order));
    }
    if (!order.IsValid)
    {
        Console.WriteLine("Invalid order.");
        return;
    }
    // Main processing logic here
    Console.WriteLine("Processing order...");
}
```

<Bad Code>

```
// Bad Example: Deeply nested code
```

```
void ProcessOrder(Order order)
{
   if (order != null)
   {
      if (order.IsValid)
      {
         // Main processing logic here
         Console.WriteLine("Processing order...");
      }
      else
      {
         Console.WriteLine("Invalid order.");
      }
   }
   else
   {
      throw new ArgumentNullException(nameof(order));
   }
}
```

Using guard clauses, as shown in the good example, allows you to handle error conditions early and exit the method if necessary. This keeps the main logic at the main level of the method, making it easier to read and maintain. In contrast, the bad example demonstrates deep nesting, which makes the code harder to follow and increases cognitive load.

<Memo>
Guard clauses help to adhere to the "Fail Fast" principle, which suggests that a program should immediately report any condition that is likely to indicate a failure. This helps in identifying issues early and improves overall code quality.

43

Use switch statements for multiple conditions.

Utilize switch statements to handle multiple conditions efficiently, improving readability and maintainability.

When dealing with multiple conditional branches, switch statements can simplify the code, making it easier to read and manage compared to multiple if-else statements.

< Good Code >

```
public string GetDayName(int day)
{
    // Use switch statement for better readability
    switch (day)
    {
        case 1:
            return "Monday";
        case 2:
            return "Tuesday";
        case 3:
            return "Wednesday";
        case 4:
            return "Thursday";
        case 5:
            return "Friday";
        case 6:
            return "Saturday";
        case 7:
            return "Sunday";
        default:
```

```
        return "Invalid day";
    }
}
```

<Bad Code>

```
public string GetDayName(int day)
{
    // Multiple if-else statements make the code harder to read
    if (day == 1)
    {
        return "Monday";
    }
    else if (day == 2)
    {
        return "Tuesday";
    }
    else if (day == 3)
    {
        return "Wednesday";
    }
    else if (day == 4)
    {
        return "Thursday";
    }
    else if (day == 5)
    {
        return "Friday";
    }
    else if (day == 6)
    {
        return "Saturday";
    }
    else if (day == 7)
    {
        return "Sunday";
    }
```

```
    else
    {
        return "Invalid day";
    }
}
```

Using a switch statement in the good example makes it clear and straightforward to see all possible conditions for the day variable. It avoids the clutter and complexity of multiple if-else statements, which can be harder to read and maintain. This approach is more scalable and organized.

<Memo>
The switch statement in C# can also be used with string types starting from C# 7.0, adding flexibility in handling multiple conditions.

44

Break long methods into smaller, more manageable ones.

Divide long methods into smaller, focused methods to enhance readability and maintainability.

Long methods can be difficult to understand and maintain. Breaking them into smaller methods helps keep the code organized and easier to manage, promoting code reuse and better debugging.

< Good Code >

```
public void ProcessOrder(Order order)
{
   // Split tasks into smaller methods for clarity
   ValidateOrder(order);
   CalculateTotal(order);
   ProcessPayment(order);
   GenerateReceipt(order);
}
private void ValidateOrder(Order order)
{
   // Validation logic here
}
private void CalculateTotal(Order order)
{
   // Calculation logic here
}
private void ProcessPayment(Order order)
{
   // Payment processing logic here
}
```

```
private void GenerateReceipt(Order order)
{
    // Receipt generation logic here
}
```

<Bad Code>

```
public void ProcessOrder(Order order)
{
    // Long method with multiple responsibilities
    if (order == null)
    {
        throw new ArgumentNullException(nameof(order));
    }
    // Validate order
    if (order.Items.Count == 0)
    {
        throw new InvalidOperationException("Order must have at least one
item.");
    }
    // Calculate total
    decimal total = 0;
    foreach (var item in order.Items)
    {
        total += item.Price;
    }
    order.Total = total;
    // Process payment
    PaymentService paymentService = new PaymentService();
    paymentService.Process(order);
    // Generate receipt
    Receipt receipt = new Receipt();
    receipt.OrderId = order.Id;
    receipt.Total = order.Total;
    // More receipt generation logic
}
```

The good example breaks down the ProcessOrder method into smaller, single-responsibility methods. This separation of concerns makes the code easier to read, test, and maintain. The bad example combines all logic into one long method, making it harder to understand and manage.

<Memo>
Following the Single Responsibility Principle (SRP) from SOLID principles helps in creating methods that focus on one task, improving code quality and maintainability.

45

Use foreach instead of for when iterating over collections.

The foreach loop simplifies iteration over collections, making code more readable and less error-prone.

Using foreach improves readability and reduces the likelihood of errors compared to the traditional for loop.

< Good Code >

```
// Iterating over a list of strings using foreach
List<string> names = new List<string> { "Alice", "Bob", "Charlie" };
foreach (string name in names)
{
   Console.WriteLine(name);
}
```

<Bad Code>

```
// Iterating over a list of strings using for
List<string> names = new List<string> { "Alice", "Bob", "Charlie" };
for (int i = 0; i < names.Count; i++)
{
   Console.WriteLine(names[i]);
}
```

The foreach loop automatically handles the iteration over each element in the collection without needing to manage the index manually. This makes the code less prone to off-by-one errors and easier to read. The for loop,

while more flexible, requires manual handling of the index and can introduce bugs if not managed correctly.

<Memo>
The foreach loop in C# is syntactic sugar for using the enumerator pattern, which internally uses the IEnumerator interface to iterate over a collection.

46

Use try-catch blocks to handle exceptions gracefully.

Try-catch blocks allow you to handle exceptions gracefully, preventing the application from crashing and providing meaningful error messages.

Implementing try-catch blocks ensures that your application can handle runtime errors without unexpected crashes, improving reliability and user experience.

< Good Code >

```
// Handling a possible exception during file reading
try
{
    string content = File.ReadAllText("example.txt");
    Console.WriteLine(content);
}
catch (FileNotFoundException ex)
{
    Console.WriteLine("File not found: " + ex.Message);
}
catch (UnauthorizedAccessException ex)
{
    Console.WriteLine("Access denied: " + ex.Message);
}
catch (Exception ex)
{
    Console.WriteLine("An error occurred: " + ex.Message);
}
```

<Bad Code>

```
// Not handling exceptions, leading to a potential crash
string content = File.ReadAllText("example.txt");
Console.WriteLine(content);
```

Using try-catch blocks around code that can throw exceptions ensures that your application can handle errors gracefully. In the good example, specific exceptions are caught, and appropriate messages are displayed. The bad example lacks error handling, which can lead to unhandled exceptions and application crashes, leaving the user without guidance on what went wrong.

<Memo>

In C#, it's good practice to catch specific exceptions before catching the general Exception type to handle known error conditions appropriately and avoid masking unexpected issues.

47

Avoid empty catch blocks; at least log the exception.

Empty catch blocks can hide errors and make debugging difficult. Always log exceptions to understand what went wrong.

Empty catch blocks are a common mistake that can lead to silent failures. By logging exceptions, you ensure that errors are recorded and can be addressed.

< Good Code >

```
csharptry
{
    // Code that may throw an exception
}
catch (Exception ex)
{
    // Log the exception
    Console.WriteLine($"An error occurred: {ex.Message}");
}
```

<Bad Code>

```
csharptry
{
    // Code that may throw an exception
}
catch (Exception)
{
    // Empty catch block
```

```
}
```

In the good example, the exception is caught and logged, providing valuable information for debugging. In the bad example, the empty catch block swallows the exception, making it difficult to diagnose issues.

<Memo>
Logging frameworks like NLog or log4net can be used to log exceptions to various outputs, such as files, databases, or remote servers, providing more flexibility and control over logging.

48

Use finally blocks to clean up resources.

Finally blocks ensure that resources are released properly, even if an exception occurs.

Finally blocks are used to execute code that must run regardless of whether an exception is thrown, such as closing files or releasing network resources.

< Good Code >

```
csharpFileStream fileStream = null;
try
{
    fileStream = new FileStream("example.txt", FileMode.Open);
    // Code that works with the file
}
catch (Exception ex)
{
    // Log the exception
    Console.WriteLine($"An error occurred: {ex.Message}");
}
finally
{
    // Ensure the file is closed
    if (fileStream != null)
    {
        fileStream.Close();
    }
}
```

<Bad Code>

```
csharpFileStream fileStream = null;
try
{
    fileStream = new FileStream("example.txt", FileMode.Open);
    // Code that works with the file
}
catch (Exception ex)
{
    // Log the exception
    Console.WriteLine($"An error occurred: {ex.Message}");
}
// No finally block to ensure the file is closed
```

In the good example, the finally block ensures that the file stream is closed, preventing resource leaks. In the bad example, if an exception occurs, the file stream may not be closed, leading to potential resource leaks.

<Memo>
The using statement in C# can be used as a shorthand for try-finally blocks when working with objects that implement the IDisposable interface, ensuring that resources are automatically cleaned up.

49

Throw specific exceptions instead of generic ones.

Using specific exceptions makes error handling more precise and easier to debug.

Throwing specific exceptions instead of generic ones helps provide clearer information about what went wrong.

< Good Code >

```
// Good example: Specific exception
public void OpenFile(string fileName)
{
    if (string.IsNullOrEmpty(fileName))
    {
        throw new ArgumentException("File name cannot be null or empty",
nameof(fileName));
    }
    // Additional code to open the file
}
```

<Bad Code>

```
// Bad example: Generic exception
public void OpenFile(string fileName)
{
    if (string.IsNullOrEmpty(fileName))
    {
        throw new Exception("Something went wrong");
    }
```

```
    // Additional code to open the file
}
```

In the good example, ArgumentException is thrown with a specific message, making it clear that the error is due to an invalid argument. In the bad example, a generic Exception is thrown, providing no useful information about the nature of the error.

<Memo>
Using specific exceptions can improve maintainability and debugging by allowing developers to catch and handle different types of errors appropriately.

50

Use using statements to ensure proper disposal of resources.

The using statement ensures that disposable resources are properly released.

Using the using statement helps manage the lifecycle of resources that implement IDisposable, ensuring they are correctly disposed of even if an exception occurs.

< Good Code >

```
// Good example: Using statement
public void ReadFile(string filePath)
{
    using (StreamReader reader = new StreamReader(filePath))
    {
        string content = reader.ReadToEnd();
        Console.WriteLine(content);
    } // StreamReader is automatically disposed here
}
```

<Bad Code>

```
// Bad example: Manual disposal
public void ReadFile(string filePath)
{
    StreamReader reader = new StreamReader(filePath);
    try
    {
        string content = reader.ReadToEnd();
        Console.WriteLine(content);
```

```
    }
    finally
    {
        if (reader != null)
        {
            reader.Dispose();
        }
    }
}
```

In the good example, the using statement ensures that the StreamReader is disposed of automatically when it goes out of scope. In the bad example, the disposal is done manually in a finally block, which is more error-prone and verbose.

<Memo>
The using statement in C# is syntactic sugar that simplifies the usage of the try-finally pattern for disposing of resources.

51

Use const for values that never change.

Declare constant values using the const keyword to improve code readability and maintainability.

Using const for values that never change ensures that these values are easily identifiable and prevents accidental modification.

< Good Code >

```
public class Circle
{
    // Good example: using const for a fixed value
    private const double Pi = 3.14159;
    public double CalculateCircumference(double radius)
    {
        return 2 * Pi * radius; // Using the constant value Pi
    }
}
```

<Bad Code>

```
public class Circle
{
    // Bad example: not using const for a fixed value
    private double pi = 3.14159;
    public double CalculateCircumference(double radius)
    {
        return 2 * pi * radius; // Using a non-constant value
    }
}
```

Using const makes it clear that Pi is a constant value that will not change, which enhances readability and prevents accidental changes. In the bad example, pi is not marked as constant, making it possible to modify it, which could lead to unexpected behavior.

<Memo>
In C#, const fields are implicitly static and must be initialized with a constant expression at the time of declaration. They cannot be changed thereafter.

52

Declare variables as close to their usage as possible.

Define variables at the point of first use to enhance code clarity and reduce the scope of the variables.

Declaring variables close to their usage helps in understanding the purpose of the variable and limits its scope, making the code easier to read and maintain.

< Good Code >

```
public void ProcessData()
{
    // Good example: declaring variables near their usage
    for (int i = 0; i < 10; i++)
    {
        int result = i * 2; // Variable declared close to its usage
        Console.WriteLine(result);
    }
}
```

<Bad Code>

```
public void ProcessData()
{
    // Bad example: declaring variables far from their usage
    int result;
    for (int i = 0; i < 10; i++)
    {
        result = i * 2; // Variable used far from its declaration
```

```
        Console.WriteLine(result);
    }
}
```

In the good example, the variable result is declared right where it is needed, making the code more readable and its scope limited to the loop. In the bad example, result is declared at the beginning of the method, which can cause confusion and potential misuse as its purpose is not immediately clear.

<Memo>
Reducing the scope of variables by declaring them close to their usage can help in avoiding errors related to variable reuse and makes the code easier to understand for others.

53

Use meaningful names for all variables

Give variables descriptive names that reflect their purpose and content.

When declaring variables, choose names that clearly convey their meaning and role in the code.

< Good Code >

```
csharpint currentYear = DateTime.Now.Year; // Descriptive name
double salesTax = 0.07; // Meaningful name
string customerName = "John Doe"; // Clear variable purpose
```

<Bad Code>

```
csharpint x = DateTime.Now.Year; // Unclear variable purpose
double t = 0.07; // Ambiguous name
string s = "John Doe"; // Meaningless name
```

Using meaningful names makes the code more readable and easier to understand, especially for other developers or when revisiting the code after some time. It helps convey the intent and purpose of each variable, reducing the cognitive load required to comprehend the code.

<Memo>
Many coding style guides, such as Microsoft's .NET Framework Design Guidelines, recommend using descriptive and meaningful names for all identifiers, including variables.

54

Avoid using the same variable for multiple purposes

Assign a single responsibility to each variable and avoid reusing it for different purposes.

Variables should have a clear and consistent role throughout their scope. Reusing the same variable for multiple purposes can lead to confusion and potential bugs.

< Good Code >

```
csharpint age = 25;
double taxRate = 0.08;
string fullName = "Jane Smith";
// Calculating tax
double taxAmount = purchaseAmount * taxRate;
```

<Bad Code>

```
csharpint value = 25; // Initially age
double rate = 0.08; // Tax rate
string name = "Jane Smith"; // Full name
// Later, value is reused for a different purpose
value = 1000; // Now represents a purchase amount
double taxAmount = value * rate; // Mixing age and purchase amount
```

Reusing variables for different purposes can make the code harder to read, understand, and maintain. It can also introduce subtle bugs if the variable's value is unintentionally overwritten or used in an unexpected context. By

assigning a single responsibility to each variable, the code becomes more self-documenting and less prone to errors.

<Memo>
The principle of "single responsibility" is a fundamental concept in software design and is often applied not only to variables but also to classes, functions, and other code constructs.

55

Use var when the type is clear from the context.

Using var can make your code cleaner and more readable when the type is obvious from the context.

When the type of a variable is clear from the right-hand side of the assignment, using var can reduce redundancy and improve readability.

< Good Code >

```
csharpvar customer = new Customer(); // The type is clear from the context
var totalAmount = 100.0; // The type is clear from the context
```

<Bad Code>

```
csharpCustomer customer = new Customer(); // Redundant type declaration
double totalAmount = 100.0; // Redundant type declaration
```

In the good example, var is used because the type of the variable is evident from the right-hand side of the assignment. This reduces redundancy and makes the code cleaner. In the bad example, the type is explicitly declared, which is unnecessary and makes the code more verbose.

<Memo>
The var keyword was introduced in C# 3.0 as part of the language's support for implicitly typed local variables.

56

Limit the scope of variables to the smallest possible.

Declare variables in the smallest scope possible to improve readability and maintainability.

Limiting the scope of variables to the smallest possible context helps prevent errors and makes the code easier to understand and maintain.

< Good Code >

```
csharppublic void ProcessOrder()
{
    if (order.IsValid)
    {
        var discount = CalculateDiscount(order); // Variable declared in the smallest scope
        ApplyDiscount(order, discount);
    }
}
```

<Bad Code>

```
csharppublic void ProcessOrder()
{
    double discount; // Variable declared in a larger scope than necessary
    if (order.IsValid)
    {
        discount = CalculateDiscount(order);
        ApplyDiscount(order, discount);
    }
```

```
}
```

In the good example, the discount variable is declared within the if block, limiting its scope to where it is needed. This makes the code easier to read and reduces the risk of errors. In the bad example, discount is declared at the beginning of the method, giving it a broader scope than necessary, which can lead to potential misuse or errors.

<Memo>
Limiting the scope of variables is a principle of good software design known as "encapsulation," which helps in managing complexity and improving code quality.

57

Use descriptive names for loop counters.

Using descriptive names for loop counters improves code readability and maintainability.

When writing loops, using generic names like i, j, or k can make the code harder to understand. Instead, use descriptive names that indicate the purpose of the loop counter.

< Good Code >

```
csharp// Loop through each student in the list
for (int studentIndex = 0; studentIndex < students.Count; studentIndex++)
{
    Console.WriteLine(students[studentIndex].Name);
}
```

<Bad Code>

```
csharp// Loop through each student in the list
for (int i = 0; i < students.Count; i++)
{
    Console.WriteLine(students[i].Name);
}
```

In the good example, studentIndex clearly indicates that the loop counter is used to index students. This makes the code more understandable at a glance. In the bad example, i is less informative, requiring the reader to infer its purpose from the context.

<Memo>

Descriptive names are not only useful for loop counters but also for variables, methods, and classes. They help in making the code self-documenting, reducing the need for excessive comments.

58

Avoid global variables; use class fields or properties instead.

Using class fields or properties instead of global variables enhances encapsulation and reduces potential side effects.

Global variables can be accessed and modified from anywhere in the code, making it difficult to track changes and debug issues. Using class fields or properties confines the scope of variables, improving code structure and reliability.

< Good Code >

```
csharppublic class Student
{
    // Encapsulated field
    private string name;
    // Property to access the field
    public string Name
    {
        get { return name; }
        set { name = value; }
    }
    public void PrintName()
    {
        Console.WriteLine(Name);
    }
}
// Usage
Student student = new Student();
student.Name = "John Doe";
student.PrintName();
```

<Bad Code>

```
csharp// Global variable
string name;
void PrintName()
{
    Console.WriteLine(name);
}
// Usage
name = "John Doe";
PrintName();
```

In the good example, the name field is encapsulated within the Student class, and access is controlled through the Name property. This approach ensures that the variable is only modified in a controlled manner. In the bad example, the global variable name can be modified from anywhere, leading to potential bugs and making the code harder to maintain.

<Memo>

Encapsulation is a fundamental principle of object-oriented programming (OOP). It helps in bundling the data with the methods that operate on the data, restricting direct access to some of the object's components.

59

Use readonly for variables that should not change after initialization.

Using readonly ensures that variables are only assigned once, improving code reliability and readability.

Declaring variables as readonly when they should not change after initialization helps prevent accidental modifications and makes the code easier to understand.

< Good Code >

```
csharppublic class Example
{
   // This variable is readonly and cannot be changed after initialization
   private readonly int maxItems;
   public Example(int maxItems)
   {
      this.maxItems = maxItems; // Initialization in the constructor
   }
   public void DisplayMaxItems()
   {
      Console.WriteLine($"Max items: {maxItems}");
   }
}
```

<Bad Code>

```
csharppublic class Example
{
   // This variable is not readonly and can be changed anywhere in the class
```

```
    private int maxItems;
    public Example(int maxItems)
    {
        this.maxItems = maxItems;
    }
    public void UpdateMaxItems(int newMax)
    {
        maxItems = newMax; // Potentially dangerous modification
    }
    public void DisplayMaxItems()
    {
        Console.WriteLine($"Max items: {maxItems}");
    }
}
```

In the good example, the maxItems variable is declared as readonly, ensuring it can only be set once, either at the point of declaration or within the constructor. This prevents accidental changes to the variable, making the code more predictable and easier to maintain. In the bad example, maxItems can be modified anywhere in the class, which can lead to unintended side effects and bugs.

<Memo>
The readonly keyword in C# is similar to the final keyword in Java and the const keyword in C++, but readonly allows for initialization in the constructor, providing more flexibility.

60

Initialize variables where they are declared.

Initializing variables at the point of declaration improves code clarity and reduces the risk of uninitialized variables.

Initializing variables where they are declared helps ensure that they are always in a valid state and makes the code easier to read and maintain.

< Good Code >

```
csharppublic class Example
{
    // Variable is initialized at the point of declaration
    private int maxItems = 10;
    public void DisplayMaxItems()
    {
        Console.WriteLine($"Max items: {maxItems}");
    }
}
```

<Bad Code>

```
csharppublic class Example
{
    // Variable is declared without initialization
    private int maxItems;
    public Example()
    {
        maxItems = 10; // Initialization happens later
    }
```

```
    public void DisplayMaxItems()
    {
        Console.WriteLine($"Max items: {maxItems}");
    }
}
```

In the good example, the maxItems variable is initialized at the point of declaration, ensuring it always has a valid value. This makes the code more readable and reduces the risk of using an uninitialized variable. In the bad example, maxItems is declared without initialization, and its value is set later in the constructor. This can lead to potential issues if the variable is accessed before it is initialized.

<Memo>
Initializing variables at the point of declaration is a common practice in many programming languages, including C#, Java, and Python, as it promotes better coding habits and reduces the likelihood of runtime errors.

61

Ensure each method performs a single, well-defined task.

Write methods that focus on a single responsibility to improve readability and maintainability.

A method should do one thing and do it well. This makes the code easier to understand, test, and maintain.

< Good Code >

```
csharp// Good example: Method with a single responsibility
public class UserService
{
    public bool IsUserValid(User user)
    {
        return user != null && !string.IsNullOrEmpty(user.Name);
    }
}
```

<Bad Code>

```
csharp// Bad example: Method with multiple responsibilities
public class UserService
{
    public bool IsUserValid(User user)
    {
        if (user == null)
        {
            return false;
        }
```

```
        if (string.IsNullOrEmpty(user.Name))
        {
            return false;
        }
        LogUserValidation(user);
        return true;
    }
    private void LogUserValidation(User user)
    {
        // Log user validation
    }
}
```

In the good example, the IsUserValid method only checks if the user is valid, adhering to the single responsibility principle. In the bad example, the method also logs the validation, which should be a separate concern.

<Memo>
The Single Responsibility Principle (SRP) is one of the SOLID principles of object-oriented design, which states that a class or method should have only one reason to change.

62

Avoid long methods; break them into smaller ones.

Refactor long methods into smaller, more manageable ones to enhance readability and maintainability.

Long methods can be difficult to read and understand. Breaking them into smaller methods makes the code more modular and easier to manage.

< Good Code >

```
csharp// Good example: Short, focused methods
public class OrderService
{
    public void ProcessOrder(Order order)
    {
        ValidateOrder(order);
        CalculateTotal(order);
        SaveOrder(order);
    }
    private void ValidateOrder(Order order)
    {
        // Validation logic
    }
    private void CalculateTotal(Order order)
    {
        // Calculation logic
    }
    private void SaveOrder(Order order)
    {
        // Save logic
    }
```

```
}
```

<Bad Code>

```
csharp// Bad example: Long method with multiple responsibilities
public class OrderService
{
    public void ProcessOrder(Order order)
    {
        // Validation logic
        if (order == null || order.Items.Count == 0)
        {
            throw new ArgumentException("Invalid order");
        }
        // Calculation logic
        decimal total = 0;
        foreach (var item in order.Items)
        {
            total += item.Price * item.Quantity;
        }
        order.Total = total;
        // Save logic
        // Code to save the order to the database
    }
}
```

In the good example, the ProcessOrder method is broken down into smaller methods, each handling a specific task. This makes the code easier to read and maintain. In the bad example, the ProcessOrder method is long and handles multiple tasks, making it harder to understand and maintain.

<Memo>

Refactoring long methods into smaller ones is a common practice in clean code principles, which helps in achieving better code modularity and reusability.

63

Use helper methods to encapsulate complex logic.

Encapsulate complex logic into helper methods to improve code readability and maintainability.

Breaking down complex logic into smaller, reusable helper methods makes the code easier to understand and maintain.

< Good Code >

```
csharppublic class OrderProcessor
{
    public void ProcessOrder(Order order)
    {
        if (IsOrderValid(order))
        {
            CalculateOrderTotal(order);
            SaveOrder(order);
        }
    }
    private bool IsOrderValid(Order order)
    {
        // Complex validation logic
        return order != null && order.Items.Count > 0;
    }
    private void CalculateOrderTotal(Order order)
    {
        // Complex calculation logic
        order.Total = order.Items.Sum(item => item.Price * item.Quantity);
    }
    private void SaveOrder(Order order)
```

```
    {
        // Complex saving logic
        // Save order to database
    }
}
```

<Bad Code>

```
csharppublic class OrderProcessor
{
    public void ProcessOrder(Order order)
    {
        if (order != null && order.Items.Count > 0)
        {
            order.Total = 0;
            foreach (var item in order.Items)
            {
                order.Total += item.Price * item.Quantity;
            }
            // Save order to database
        }
    }
}
```

In the good example, the complex logic is encapsulated into helper methods (IsOrderValid, CalculateOrderTotal, and SaveOrder), making the ProcessOrder method cleaner and easier to read. In the bad example, all the logic is crammed into the ProcessOrder method, making it harder to understand and maintain.

<Memo>

Encapsulating logic into helper methods not only improves readability but also promotes code reuse and easier unit testing.

64

Refactor code to eliminate duplication.

Refactor your code to remove duplication and improve maintainability.

Duplicated code can lead to maintenance issues and bugs. Refactoring to eliminate duplication makes the codebase cleaner and more reliable.

< Good Code >

```
csharppublic class DiscountCalculator
{
    public decimal ApplyDiscount(Order order, decimal discountRate)
    {
        return CalculateDiscount(order.Total, discountRate);
    }
    public decimal ApplySpecialDiscount(Order order, decimal
specialDiscountRate)
    {
        return CalculateDiscount(order.Total, specialDiscountRate);
    }
    private decimal CalculateDiscount(decimal total, decimal discountRate)
    {
        return total - (total * discountRate);
    }
}
```

<Bad Code>

```
csharppublic class DiscountCalculator
{
    public decimal ApplyDiscount(Order order, decimal discountRate)
    {
```

```
        return order.Total - (order.Total * discountRate);
    }
    public decimal ApplySpecialDiscount(Order order, decimal
specialDiscountRate)
    {
        return order.Total - (order.Total * specialDiscountRate);
    }
}
```

In the good example, the CalculateDiscount method is used to eliminate duplicated discount calculation logic. This makes the code easier to maintain and reduces the risk of errors. In the bad example, the discount calculation logic is duplicated in both ApplyDiscount and ApplySpecialDiscount methods, making the code harder to maintain and more error-prone.

<Memo>
Refactoring to eliminate duplication is a core principle of the DRY (Don't Repeat Yourself) principle, which aims to reduce repetition of code patterns and improve overall code quality.

65

Use meaningful method names that describe their purpose.

Choose method names that clearly describe what the method does.

Using meaningful method names helps other developers understand the purpose and functionality of the method without needing to read through its implementation.

< Good Code >

```
csharp// Good example: Method name clearly describes its purpose
public class UserService
{
    public void CreateUser(string username, string password)
    {
        // Code to create a new user
    }
}
```

<Bad Code>

```
csharp// Bad example: Method name is vague and unclear
public class UserService
{
    public void DoSomething(string a, string b)
    {
        // Code to create a new user
    }
}
```

In the good example, the method name CreateUser clearly indicates that the method is responsible for creating a new user. In the bad example, the method name DoSomething is vague and does not provide any information about what the method does, making it harder for other developers to understand its purpose.

<Memo>
Meaningful method names are part of the broader concept of self-documenting code, which aims to make code more readable and maintainable without relying heavily on external documentation.

66

Use async and await for asynchronous operations.

Utilize async and await keywords to handle asynchronous operations in a clear and efficient manner.

Using async and await makes asynchronous code easier to read and maintain by avoiding complex callback patterns and improving error handling.

< Good Code >

```
csharp// Good example: Using async and await for asynchronous operations
public async Task<string> FetchDataAsync(string url)
{
    using (HttpClient client = new HttpClient())
    {
        HttpResponseMessage response = await client.GetAsync(url);
        response.EnsureSuccessStatusCode();
        return await response.Content.ReadAsStringAsync();
    }
}
```

<Bad Code>

```
csharp// Bad example: Using callbacks for asynchronous operations
public void FetchData(string url, Action<string> callback)
{
    HttpClient client = new HttpClient();
    client.GetAsync(url).ContinueWith(responseTask =>
    {
```

```
        HttpResponseMessage response = responseTask.Result;
        response.EnsureSuccessStatusCode();
        response.Content.ReadAsStringAsync().ContinueWith(readTask =>
        {
            callback(readTask.Result);
        });
    });
}
```

In the good example, the async and await keywords are used to handle asynchronous operations in a straightforward manner, making the code easier to read and maintain. In the bad example, callbacks are used, which can lead to more complex and harder-to-read code, often referred to as "callback hell."

<Memo>
The async and await keywords were introduced in C# 5.0 to simplify asynchronous programming and improve code readability by allowing developers to write asynchronous code that looks similar to synchronous code.

67

Leverage LINQ for querying and manipulating collections.

Use LINQ to simplify querying and manipulating collections in a readable and efficient manner.

LINQ (Language Integrated Query) allows for concise and readable code when dealing with collections, making it easier to perform operations like filtering, sorting, and projecting data.

< Good Code >

```
// Using LINQ to filter and sort a list of integers
var numbers = new List<int> { 5, 2, 8, 1, 4 };
var evenNumbers = numbers.Where(n => n % 2 == 0).OrderBy(n =>
n).ToList();
// evenNumbers now contains [2, 4, 8]
```

<Bad Code>

```
// Manual filtering and sorting of a list of integers
var numbers = new List<int> { 5, 2, 8, 1, 4 };
var evenNumbers = new List<int>();
foreach (var number in numbers)
{
    if (number % 2 == 0)
    {
        evenNumbers.Add(number);
    }
}
evenNumbers.Sort();
```

```
// evenNumbers now contains [2, 4, 8]
```

Using LINQ, the good example achieves filtering and sorting in a single, readable line of code. In contrast, the bad example requires multiple steps, making it more error-prone and harder to read. LINQ enhances code readability and maintainability.

<Memo>
LINQ stands for Language Integrated Query and was introduced in .NET Framework 3.5. It provides a consistent model for working with data across various sources and formats.

68

Use yield return for implementing custom iterators.

Use yield return to simplify the creation of custom iterators for collections.

The yield return statement allows you to define custom iteration logic without the need to create a temporary collection or manage an explicit state machine.

< Good Code >

```
// Implementing a custom iterator using yield return
public IEnumerable<int> GetEvenNumbers(int max)
{
    for (int i = 0; i <= max; i++)
    {
        if (i % 2 == 0)
        {
            yield return i;
        }
    }
}
// Usage: var evens = GetEvenNumbers(10); // returns 0, 2, 4, 6, 8, 10
```

<Bad Code>

```
// Implementing a custom iterator using a temporary list
public List<int> GetEvenNumbers(int max)
{
    var evens = new List<int>();
    for (int i = 0; i <= max; i++)
```

```
    {
        if (i % 2 == 0)
        {
            evens.Add(i);
        }
    }
    return evens;
}
// Usage: var evens = GetEvenNumbers(10); // returns 0, 2, 4, 6, 8, 10
```

The good example uses yield return to produce even numbers on-the-fly without storing them in a temporary list, making the code more efficient and easier to read. The bad example creates a temporary list to store results, which is less efficient and more cumbersome.

<Memo>
The yield keyword in C# was introduced in C# 2.0, allowing developers to implement stateful iterators in a straightforward manner without needing to maintain complex state manually.

69

Utilize Task and Task<T> for parallel programming.

Use Task and Task<T> to perform parallel programming efficiently in C#.

Task and Task<T> are part of the Task Parallel Library (TPL) in C#. They provide a simple way to run code asynchronously and handle parallel operations.

< Good Code >

```
csharpusing System;
using System.Threading.Tasks;
class Program
{
   static async Task Main(string[] args)
   {
      Task<int> task1 = Task.Run(() => Compute(10));
      Task<int> task2 = Task.Run(() => Compute(20));
      int result1 = await task1;
      int result2 = await task2;
      Console.WriteLine($"Result1: {result1}, Result2: {result2}");
   }
   static int Compute(int value)
   {
      // Simulate a time-consuming operation
      Task.Delay(1000).Wait();
      return value * value;
   }
}
```

<Bad Code>

```
csharpusing System;
class Program
{
    static void Main(string[] args)
    {
        int result1 = Compute(10);
        int result2 = Compute(20);
        Console.WriteLine($"Result1: {result1}, Result2: {result2}");
    }
    static int Compute(int value)
    {
        // Simulate a time-consuming operation
        System.Threading.Thread.Sleep(1000);
        return value * value;
    }
}
```

In the good example, Task.Run is used to run Compute method asynchronously, allowing both tasks to run in parallel. The await keyword ensures that the main method waits for both tasks to complete before printing the results. In the bad example, the Compute method is called synchronously, causing the operations to run sequentially and blocking the main thread.

<Memo>

The Task Parallel Library (TPL) was introduced in .NET Framework 4.0 to simplify parallel programming and improve performance by utilizing multiple cores of the CPU.

70

Use lock statements to handle concurrency issues.

Use lock statements to manage access to shared resources and prevent race conditions in multithreaded applications.

The lock statement in C# is used to ensure that a block of code runs to completion without interruption by other threads, thus preventing data corruption and ensuring thread safety.

< Good Code >

```
csharpusing System;
using System.Threading;
class Program
{
   private static readonly object _lock = new object();
   private static int _counter = 0;
   static void Main(string[] args)
   {
      Thread thread1 = new Thread(IncrementCounter);
      Thread thread2 = new Thread(IncrementCounter);
      thread1.Start();
      thread2.Start();
      thread1.Join();
      thread2.Join();
      Console.WriteLine($"Final Counter: {_counter}");
   }
   static void IncrementCounter()
   {
      for (int i = 0; i < 1000; i++)
      {
```

```
            lock (_lock)
            {
                _counter++;
            }
        }
    }
}
```

<Bad Code>

```
csharpusing System;
using System.Threading;
class Program
{
    private static int _counter = 0;
    static void Main(string[] args)
    {
        Thread thread1 = new Thread(IncrementCounter);
        Thread thread2 = new Thread(IncrementCounter);
        thread1.Start();
        thread2.Start();
        thread1.Join();
        thread2.Join();
        Console.WriteLine($"Final Counter: {_counter}");
    }
    static void IncrementCounter()
    {
        for (int i = 0; i < 1000; i++)
        {
            _counter++;
        }
    }
}
```

In the good example, the lock statement ensures that only one thread can increment the _counter variable at a time, preventing race conditions. In the bad example, the lack of synchronization allows multiple threads to increment _counter simultaneously, leading to unpredictable results and potential data corruption.

<Memo>
The lock statement is syntactic sugar for Monitor.Enter and Monitor.Exit, providing a simpler way to ensure mutual exclusion in critical sections of code.

71

Break down complex expressions into multiple lines.

Splitting complex expressions into multiple lines enhances readability and maintainability.

Complex expressions can be hard to read and understand at a glance. Breaking them down into multiple lines makes the code more readable and easier to debug.

< Good Code >

```
// Breaking down a complex expression into multiple lines
int result = (a + b) * c;
result = result / d;
result = result - e;
```

<Bad Code>

```
// Combining all operations in a single line
int result = ((a + b) * c / d) - e;
```

In the good code example, each operation is clearly separated, making it easier to understand the order of operations and debug if needed. The bad code example combines all operations in a single line, making it harder to follow and increasing the risk of mistakes.

<Memo>
The concept of breaking down complex expressions into simpler steps is often referred to as "stepwise refinement," which is a core principle in

computer programming to improve clarity and maintainability.

72

Use intermediate variables to store results of sub-expressions.

Using intermediate variables helps clarify the purpose of each part of an expression.

Storing the results of sub-expressions in intermediate variables improves code readability and allows for easier debugging and testing of individual components.

< Good Code >

```
// Using intermediate variables for clarity
int sum = a + b;
int product = sum * c;
int quotient = product / d;
int result = quotient - e;
```

<Bad Code>

```
// Performing all calculations in a single line
int result = (((a + b) * c) / d) - e;
```

In the good code example, intermediate variables sum, product, and quotient are used to store the results of sub-expressions. This makes the code easier to read and understand. In the bad code example, all calculations are done in a single line, which can be confusing and error-prone.

<Memo>

Using intermediate variables is a common practice in clean coding and is recommended by many coding standards and guidelines, including those by organizations like Microsoft and Google.

73

Use parentheses to make the order of operations clear.

Using parentheses in expressions ensures that the order of operations is explicit and clear to anyone reading the code.

When writing complex expressions, the default order of operations can sometimes be confusing. Using parentheses helps clarify the intended order, making the code more readable and maintainable.

< Good Code >

```
csharpint result = (a + b) * (c - d); // Clear order of operations
```

<Bad Code>

```
csharpint result = a + b * c - d; // Unclear order of operations
```

In the good example, the use of parentheses makes it clear that the addition and subtraction should be performed before the multiplication. In the bad example, the order of operations is ambiguous and could lead to misunderstandings or errors.

<Memo>
The order of operations in C# follows the standard mathematical precedence: parentheses, exponents, multiplication and division, and finally addition and subtraction.

74

Avoid chaining multiple method calls in a single statement.

Chaining multiple method calls in a single statement can make the code difficult to read and debug. It's better to break them into separate statements.

While chaining method calls can make the code look concise, it often sacrifices readability and makes debugging more challenging. Breaking down the calls into separate statements improves clarity.

< Good Code >

```
csharpvar list = GetList();
var filteredList = list.Where(x => x.IsActive);
var sortedList = filteredList.OrderBy(x => x.Name);
var result = sortedList.ToList(); // Clear and easy to follow
```

<Bad Code>

```
csharpvar result = GetList().Where(x => x.IsActive).OrderBy(x =>
x.Name).ToList(); // Hard to read and debug
```

In the good example, each step of the process is broken down into a separate statement, making it clear what each step does. In the bad example, chaining all the method calls together makes it harder to understand and debug.

<Memo>
Method chaining is a common practice in fluent interfaces, but it should be used judiciously to maintain code readability and simplicity.

75

Use descriptive variable names for intermediate results.

Using descriptive variable names makes the code more readable and maintainable.

When writing code, it's important to use variable names that clearly describe their purpose, especially for intermediate results. This helps other developers understand the code quickly.

< Good Code >

```
csharp// Calculate the area of a rectangle
double length = 5.0;
double width = 3.0;
double area = length * width; // Descriptive variable name for the result
Console.WriteLine($"The area of the rectangle is {area}");
```

<Bad Code>

```
csharp// Calculate the area of a rectangle
double a = 5.0;
double b = 3.0;
double c = a * b; // Non-descriptive variable name for the result
Console.WriteLine($"The area of the rectangle is {c}");
```

In the good example, the variable area clearly indicates what the value represents, making the code easier to understand. In the bad example, the variable c is not descriptive, which can confuse readers about its purpose.

<Memo>
Descriptive variable names are part of the broader concept of self-documenting code, which aims to make code understandable without extensive comments.

76

Extract unrelated logic into separate methods.

Separating unrelated logic into different methods improves code readability and reusability.

By extracting unrelated logic into separate methods, you can make your code more modular and easier to understand. This practice also facilitates testing and maintenance.

< Good Code >

```
csharp// Main method to process data
public void ProcessData()
{
    string data = GetDataFromSource();
    string processedData = ProcessDataLogic(data);
    SaveData(processedData);
}
// Method to get data from a source
private string GetDataFromSource()
{
    // Logic to get data
    return "raw data";
}
// Method to process data
private string ProcessDataLogic(string data)
{
    // Logic to process data
    return data.ToUpper();
}
// Method to save data
```

```
private void SaveData(string data)
{
    // Logic to save data
    Console.WriteLine($"Data saved: {data}");
}
```

<Bad Code>

```
csharp// Main method to process data
public void ProcessData()
{
    // Logic to get data
    string data = "raw data";
    // Logic to process data
    string processedData = data.ToUpper();
    // Logic to save data
    Console.WriteLine($"Data saved: {processedData}");
}
```

In the good example, the logic for getting, processing, and saving data is separated into different methods, making the code more modular and easier to follow. In the bad example, all logic is combined in a single method, making it harder to read and maintain.

<Memo>

Modular code is a key principle of the SOLID design principles, particularly the Single Responsibility Principle, which states that a class or method should have only one reason to change.

77

Use helper methods to encapsulate common functionality.

Encapsulate repetitive code into helper methods to improve readability and maintainability.

When you find yourself writing the same code multiple times, it's a good practice to encapsulate that code into a helper method. This makes your code more modular, easier to read, and simpler to maintain.

< Good Code >

```
csharp// Good example: Using a helper method to encapsulate common functionality
public class Calculator
{
    public int Add(int a, int b)
    {
        return PerformOperation(a, b, (x, y) => x + y);
    }
    public int Subtract(int a, int b)
    {
        return PerformOperation(a, b, (x, y) => x - y);
    }
    private int PerformOperation(int a, int b, Func<int, int, int> operation)
    {
        return operation(a, b);
    }
}
```

<Bad Code>

```
csharp// Bad example: Repeating the same code in multiple methods
public class Calculator
{
    public int Add(int a, int b)
    {
        return a + b;
    }
    public int Subtract(int a, int b)
    {
        return a - b;
    }
}
```

In the good example, the PerformOperation helper method encapsulates the common functionality of performing an operation on two integers. This reduces code duplication and makes the Add and Subtract methods more concise and easier to understand. In the bad example, the same logic is repeated in both methods, making the code harder to maintain and more error-prone.

<Memo>
Using helper methods not only improves code readability but also makes it easier to test individual pieces of functionality, leading to more robust and reliable code.

78

Refactor large methods to improve readability.

Break down large methods into smaller, more manageable pieces to enhance readability and maintainability.

Large methods can be difficult to read, understand, and maintain. Refactoring them into smaller methods with clear, descriptive names can make your code more modular and easier to follow.

< Good Code >

```
csharp// Good example: Refactoring a large method into smaller methods
public class OrderProcessor
{
    public void ProcessOrder(Order order)
    {
        ValidateOrder(order);
        CalculateTotal(order);
        SaveOrder(order);
    }
    private void ValidateOrder(Order order)
    {
        // Validation logic here
    }
    private void CalculateTotal(Order order)
    {
        // Calculation logic here
    }
    private void SaveOrder(Order order)
    {
        // Save logic here
```

```
    }
}
```

<Bad Code>

```
csharp// Bad example: Large method with multiple responsibilities
public class OrderProcessor
{
    public void ProcessOrder(Order order)
    {
        // Validation logic here
        if (order == null)
        {
            throw new ArgumentNullException(nameof(order));
        }
        // Calculation logic here
        order.Total = order.Items.Sum(item => item.Price);
        // Save logic here
        // Assume SaveToDatabase is a method that saves the order to a
database
        SaveToDatabase(order);
    }
    private void SaveToDatabase(Order order)
    {
        // Database save logic here
    }
}
```

In the good example, the ProcessOrder method is refactored into smaller methods, each with a single responsibility: ValidateOrder, CalculateTotal, and SaveOrder. This makes the code easier to read and maintain. In the bad example, the ProcessOrder method handles validation, calculation, and saving, making it large and difficult to manage.

<Memo>

The Single Responsibility Principle (SRP) is one of the SOLID principles of object-oriented design. It states that a class or method should have only one reason to change, which is achieved by refactoring large methods into smaller ones with distinct responsibilities.

79

Use design patterns to solve common problems.

Design patterns provide reusable solutions to common problems in software design.

Using design patterns can make your code more modular, easier to understand, and maintain. Here is an example using the Singleton pattern.

< Good Code >

```
csharp// Singleton pattern example
public class Singleton
{
   private static Singleton _instance;
   private static readonly object _lock = new object();
   private Singleton() { }
   public static Singleton Instance
   {
      get
      {
         lock (_lock)
         {
            if (_instance == null)
            {
               _instance = new Singleton();
            }
            return _instance;
         }
      }
   }
   public void DoSomething()
```

```
    {
        // Method logic here
    }
}
```

<Bad Code>

```
csharp// Without Singleton pattern
public class NonSingleton
{
    public void DoSomething()
    {
        // Method logic here
    }
}
// Usage
var instance1 = new NonSingleton();
var instance2 = new NonSingleton();
instance1.DoSomething();
instance2.DoSomething();
```

The good example uses the Singleton pattern to ensure that only one instance of the class is created, which is useful for managing shared resources. The bad example creates multiple instances, which can lead to resource conflicts and is harder to manage.

<Memo>

The Singleton pattern is one of the simplest design patterns and is often used for logging, configuration settings, and managing connection pools.

80

Encapsulate complex logic within well-named methods.

Encapsulating complex logic in well-named methods improves code readability and maintainability.

Breaking down complex logic into smaller, well-named methods makes your code easier to understand and reduces the cognitive load on other developers. Here is an example.

< Good Code >

```
csharppublic class OrderProcessor
{
    public void ProcessOrder(Order order)
    {
        if (IsOrderValid(order))
        {
            CalculateOrderTotal(order);
            SaveOrder(order);
        }
    }
    private bool IsOrderValid(Order order)
    {
        // Validation logic here
        return true;
    }
    private void CalculateOrderTotal(Order order)
    {
        // Calculation logic here
    }
    private void SaveOrder(Order order)
```

```
    {
        // Save logic here
    }
}
```

<Bad Code>

```
csharppublic class OrderProcessor
{
    public void ProcessOrder(Order order)
    {
        // Complex logic all in one method
        if (order != null && order.Items.Count > 0)
        {
            decimal total = 0;
            foreach (var item in order.Items)
            {
                total += item.Price * item.Quantity;
            }
            order.Total = total;
            // Save order to database
        }
    }
}
```

The good example breaks down the order processing logic into smaller, well-named methods, making it easier to understand and maintain. The bad example has all the logic in a single method, making it harder to read and modify.

<Memo>

Encapsulation is a fundamental principle of object-oriented programming that helps in hiding the internal state and requiring all interaction to be performed through an object's methods.

81

Use async and await for asynchronous programming.

Simplify asynchronous programming by using async and await keywords.

Using async and await in C# makes asynchronous code easier to read and maintain by avoiding callback hell and making the code look more like synchronous code.

< Good Code >

```
csharppublic async Task<string> FetchDataAsync()
{
   // Asynchronously fetch data from a web service
   HttpClient client = new HttpClient();
   string result = await client.GetStringAsync("https://example.com/data");
   return result;
}
```

<Bad Code>

```
csharppublic Task<string> FetchDataAsync()
{
   HttpClient client = new HttpClient();
   return
client.GetStringAsync("https://example.com/data").ContinueWith(task =>
   {
      if (task.IsCompletedSuccessfully)
      {
         return task.Result;
      }
```

```
        else
        {
            throw task.Exception;
        }
    });
}
```

The good example uses async and await to handle asynchronous operations, making the code more readable and easier to understand. The bad example uses ContinueWith, which can lead to more complex and harder-to-read code, especially when handling exceptions.

<Memo>
The async and await keywords were introduced in C# 5.0 to simplify asynchronous programming and improve code readability.

82

Leverage LINQ for querying collections.

Use LINQ to perform complex queries on collections in a readable and concise manner.

LINQ (Language Integrated Query) allows you to write queries directly in C# to filter, sort, and manipulate collections, making the code more expressive and easier to understand.

< Good Code >

```
csharpvar numbers = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
var evenNumbers = numbers.Where(n => n % 2 == 0).ToList();
// Output: 2, 4, 6, 8, 10
foreach (var num in evenNumbers)
{
   Console.WriteLine(num);
}
```

<Bad Code>

```
csharpvar numbers = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
var evenNumbers = new List<int>();
foreach (var num in numbers)
{
   if (num % 2 == 0)
   {
      evenNumbers.Add(num);
   }
}
// Output: 2, 4, 6, 8, 10
foreach (var num in evenNumbers)
```

```
{
    Console.WriteLine(num);
}
```

The good example uses LINQ to filter the list of numbers, making the code more concise and easier to read. The bad example uses a traditional loop to achieve the same result, which is more verbose and less expressive.

<Memo>
LINQ was introduced in C# 3.0 and provides a consistent model for working with data across various sources, such as collections, databases, and XML.

83

Use using statements for resource management.

Ensure proper disposal of resources by using using statements.

The using statement in C# is a syntactic sugar for ensuring that IDisposable objects are properly disposed of, which helps in managing resources efficiently and preventing memory leaks.

< Good Code >

```
csharpusing (var stream = new FileStream("file.txt", FileMode.Open))
{
    // Use the stream
    // The stream will be automatically closed and disposed of at the end of this block
}
```

<Bad Code>

```
csharpvar stream = new FileStream("file.txt", FileMode.Open);
// Use the stream
// The stream is not disposed of properly, leading to potential memory leaks
```

In the good example, the using statement ensures that the FileStream is disposed of correctly, even if an exception occurs. In the bad example, the FileStream is not disposed of, which can lead to resource leaks and other issues.

<Memo>

The using statement can be used with any object that implements the IDisposable interface, making it a versatile tool for resource management in C#.

84

Utilize pattern matching for cleaner code.

Use pattern matching to simplify conditional logic and improve code readability.

Pattern matching in C# allows you to check an object's type and extract values in a concise and readable manner, reducing the need for complex conditional statements.

< Good Code >

```
csharpobject obj = "Hello, World!";
if (obj is string s)
{
    Console.WriteLine($"String length: {s.Length}");
}
```

<Bad Code>

```
csharpobject obj = "Hello, World!";
if (obj is string)
{
    string s = (string)obj;
    Console.WriteLine($"String length: {s.Length}");
}
```

In the good example, pattern matching is used to both check the type of obj and declare a new variable s in a single, concise statement. In the bad example, the type check and casting are done separately, making the code more verbose and harder to read.

<Memo>
Pattern matching was introduced in C# 7.0 and has been expanded in subsequent versions, providing more powerful and expressive ways to handle type checks and value extraction.

85

Use expression-bodied members for simple methods.

Simplify your code by using expression-bodied members for methods that contain a single expression.

Expression-bodied members allow you to write concise and readable code for methods that return a single value or perform a simple operation.

< Good Code >

```
csharp// Good example using expression-bodied member
public class Calculator
{
   // Single-line method using expression-bodied member
   public int Add(int a, int b) => a + b;
}
```

<Bad Code>

```
csharp// Bad example using a full method body for a simple operation
public class Calculator
{
   // Multi-line method for a simple addition
   public int Add(int a, int b)
   {
      return a + b;
   }
}
```

The good example uses an expression-bodied member to make the code more concise and readable. The bad example uses a full method body, which is unnecessary for such a simple operation. Expression-bodied members improve code readability and reduce boilerplate code.

<Memo>
Expression-bodied members were introduced in C# 6.0 and have been expanded in later versions to include constructors, destructors, and property accessors.

86

Choose the appropriate collection type for your data.

Select the most suitable collection type based on the characteristics and requirements of your data.

Using the right collection type can improve the performance, readability, and maintainability of your code. Consider factors like data size, access patterns, and required operations when choosing a collection.

< Good Code >

```
csharp// Good example using a List for dynamic data
public class StudentManager
{
    private List<string> students = new List<string>();
    public void AddStudent(string student)
    {
        students.Add(student);
    }
    public string GetStudent(int index)
    {
        return students[index];
    }
}
```

<Bad Code>

```
csharp// Bad example using an array for dynamic data
public class StudentManager
{
```

```
    private string[] students = new string[10];
    private int count = 0;
    public void AddStudent(string student)
    {
        if (count < students.Length)
        {
            students[count] = student;
            count++;
        }
        else
        {
            // Array is full, need to resize
            Array.Resize(ref students, students.Length * 2);
            students[count] = student;
            count++;
        }
    }
    public string GetStudent(int index)
    {
        return students[index];
    }
}
```

The good example uses a List<string>, which is more suitable for dynamic data as it automatically resizes and provides convenient methods for adding and accessing elements. The bad example uses an array, which requires manual resizing and additional logic to handle dynamic data, making the code more complex and error-prone.

<Memo>
The .NET framework provides various collection types such as List<T>, Dictionary<TKey, TValue>, HashSet<T>, and Queue<T>, each optimized for different scenarios and operations.

87

Use Dictionary for key-value pairs.

Dictionaries provide an efficient way to store and retrieve data using unique keys.

Using a Dictionary in C# allows you to store data in key-value pairs, making it easy to look up values based on their keys. This is particularly useful when you need fast access to data.

< Good Code >

```
csharp// Create a dictionary to store student names and their grades
Dictionary<string, int> studentGrades = new Dictionary<string, int>();
// Add some students and their grades
studentGrades.Add("Alice", 90);
studentGrades.Add("Bob", 85);
studentGrades.Add("Charlie", 88);
// Retrieve and print a student's grade
if (studentGrades.TryGetValue("Alice", out int grade))
{
    Console.WriteLine($"Alice's grade: {grade}");
}
```

<Bad Code>

```
csharp// Using arrays to store student names and grades
string[] studentNames = { "Alice", "Bob", "Charlie" };
int[] studentGrades = { 90, 85, 88 };
// Retrieve and print a student's grade
for (int i = 0; i < studentNames.Length; i++)
{
    if (studentNames[i] == "Alice")
```

```
    {
        Console.WriteLine($"Alice's grade: {studentGrades[i]}");
        break;
    }
}
```

The good example uses a Dictionary, which provides O(1) average time complexity for lookups, making it efficient and easy to read. The bad example uses parallel arrays, which require O(n) time complexity for lookups and are harder to maintain and understand.

<Memo>
Dictionaries in C# are implemented using hash tables, which allow for fast data retrieval based on keys.

88

Use List for ordered collections.

Lists provide a flexible way to store ordered collections of items.

Using a List in C# allows you to store a collection of items that can be accessed by their index. Lists are dynamic, meaning they can grow and shrink as needed, and they maintain the order of elements.

< Good Code >

```
csharp// Create a list to store student names
List<string> students = new List<string>();
// Add some students to the list
students.Add("Alice");
students.Add("Bob");
students.Add("Charlie");
// Retrieve and print all student names
foreach (string student in students)
{
    Console.WriteLine(student);
}
```

<Bad Code>

```
csharp// Using an array to store student names
string[] students = new string[3];
students[0] = "Alice";
students[1] = "Bob";
students[2] = "Charlie";
// Retrieve and print all student names
for (int i = 0; i < students.Length; i++)
{
```

```
    Console.WriteLine(students[i]);
}
```

The good example uses a List, which is more flexible and easier to work with than arrays. Lists can dynamically resize, and they provide many useful methods for manipulating the collection. The bad example uses a fixed-size array, which is less flexible and requires manual resizing if the number of elements changes.

<Memo>
Lists in C# are part of the System.Collections.Generic namespace and provide many useful methods such as Add, Remove, and Contains.

89

Use HashSet for unique elements.

HashSet ensures that all elements are unique and provides efficient operations for adding, removing, and checking for elements.

Using a HashSet in C# is a strategic way to manage collections of unique elements. It prevents duplicates and offers fast lookups.

< Good Code >

```
csharpusing System;
using System.Collections.Generic;
class Program
{
   static void Main()
   {
      // Create a HashSet to store unique elements
      HashSet<int> uniqueNumbers = new HashSet<int>();
      // Add elements to the HashSet
      uniqueNumbers.Add(1);
      uniqueNumbers.Add(2);
      uniqueNumbers.Add(3);
      uniqueNumbers.Add(1); // Duplicate, will not be added
      // Display the elements
      foreach (int number in uniqueNumbers)
      {
         Console.WriteLine(number); // Output: 1, 2, 3
      }
   }
}
```

<Bad Code>

```
csharpusing System;
using System.Collections.Generic;
class Program
{
    static void Main()
    {
        // Create a List to store elements
        List<int> numbers = new List<int>();
        // Add elements to the List
        numbers.Add(1);
        numbers.Add(2);
        numbers.Add(3);
        numbers.Add(1); // Duplicate, will be added
        // Display the elements
        foreach (int number in numbers)
        {
            Console.WriteLine(number); // Output: 1, 2, 3, 1
        }
    }
}
```

The good example uses a HashSet, which automatically handles duplicates and ensures all elements are unique. The bad example uses a List, which allows duplicates, making it less suitable for collections where uniqueness is required.

<Memo>
HashSet is part of the System.Collections.Generic namespace and is implemented using a hash table, providing average time complexity of O(1) for add, remove, and contains operations.

90

Use Queue and Stack for FIFO and LIFO collections.

Queue and Stack are specialized collections for First-In-First-Out (FIFO) and Last-In-First-Out (LIFO) operations, respectively.

In C#, Queue and Stack are used to manage collections where the order of processing elements is important. Queue processes elements in the order they were added (FIFO), while Stack processes elements in reverse order (LIFO).

< Good Code >

```
csharpusing System;
using System.Collections.Generic;
class Program
{
   static void Main()
   {
      // Queue example (FIFO)
      Queue<string> queue = new Queue<string>();
      queue.Enqueue("First");
      queue.Enqueue("Second");
      queue.Enqueue("Third");
      while (queue.Count > 0)
      {
         Console.WriteLine(queue.Dequeue()); // Output: First, Second,
Third
      }
      // Stack example (LIFO)
      Stack<string> stack = new Stack<string>();
      stack.Push("First");
```

```
        stack.Push("Second");
        stack.Push("Third");
        while (stack.Count > 0)
        {
            Console.WriteLine(stack.Pop()); // Output: Third, Second, First
        }
    }
}
```

<Bad Code>

```
csharpusing System;
using System.Collections.Generic;
class Program
{
    static void Main()
    {
        // List example used incorrectly for FIFO
        List<string> list = new List<string>();
        list.Add("First");
        list.Add("Second");
        list.Add("Third");
        while (list.Count > 0)
        {
            Console.WriteLine(list[0]);
            list.RemoveAt(0); // Output: First, Second, Third
        }
        // List example used incorrectly for LIFO
        list.Add("First");
        list.Add("Second");
        list.Add("Third");
        while (list.Count > 0)
        {
            Console.WriteLine(list[list.Count - 1]);
            list.RemoveAt(list.Count - 1); // Output: Third, Second, First
        }
    }
```

```
}
```

The good example uses Queue and Stack, which are designed for FIFO and LIFO operations, respectively. The bad example uses List, which can achieve similar results but is less efficient and not semantically appropriate for these operations.

<Memo>
Queue and Stack are part of the System.Collections.Generic namespace. Queue is implemented as a circular array, while Stack is implemented as an array, both providing O(1) time complexity for their primary operations (Enqueue/Dequeue for Queue and Push/Pop for Stack).

91

Use try-catch blocks to handle exceptions.

Encapsulate code that may throw exceptions in try-catch blocks to handle errors gracefully.

Using try-catch blocks allows you to manage runtime errors and maintain program stability by catching exceptions and providing appropriate responses.

< Good Code >

```
try
{
    // Code that may throw an exception
    int result = 10 / int.Parse(userInput);
}
catch (FormatException ex)
{
    // Handle format exception
    Console.WriteLine("Please enter a valid number.");
}
catch (DivideByZeroException ex)
{
    // Handle divide by zero exception
    Console.WriteLine("Division by zero is not allowed.");
}
```

<Bad Code>

```
// Code without try-catch block
```

```
int result = 10 / int.Parse(userInput);
Console.WriteLine("Result is " + result);
```

Good code example: The try block contains code that might throw exceptions, and the catch blocks handle specific exceptions (FormatException, DivideByZeroException). This ensures that the program can handle errors without crashing.
Bad code example: No try-catch block is used, so any exception thrown by the code (like FormatException or DivideByZeroException) will crash the program.

<Memo>
Exception handling in C# provides a way to react to exceptional circumstances (like runtime errors) in a controlled fashion, making programs more robust and easier to debug.

92

Log exceptions for debugging purposes.

Log exceptions to understand and debug issues effectively.

Logging exceptions provides insights into what went wrong, making it easier to diagnose and fix issues in the code.

< Good Code >

```
try
{
    // Code that may throw an exception
    int result = 10 / int.Parse(userInput);
}
catch (Exception ex)
{
    // Log the exception details
    File.WriteAllText("log.txt", $"Exception: {ex.Message} at
{ex.StackTrace}");
    Console.WriteLine("An error occurred. Please check the log for
details.");
}
```

<Bad Code>

```
try
{
    // Code that may throw an exception
    int result = 10 / int.Parse(userInput);
}
catch (Exception ex)
{
```

```
    // Only display a generic message
    Console.WriteLine("An error occurred.");
}
```

Good code example: The catch block logs the exception details (message and stack trace) to a file. This helps in diagnosing the issue later by reviewing the log file.
Bad code example: The catch block only displays a generic error message without logging any details, making it harder to debug and understand the cause of the issue.

<Memo>
Logging is a crucial aspect of software development, providing a way to track application behavior and errors, which is essential for maintenance and debugging in production environments.

93

Use custom exceptions for specific error conditions.

Create custom exception classes to handle specific error conditions in your application.

Using custom exceptions allows you to provide more meaningful error messages and handle specific error conditions more effectively.

< Good Code >

```
csharp// Custom exception class for a specific error condition
public class InvalidUserInputException : Exception
{
    public InvalidUserInputException(string message) : base(message) { }
}
// Method that throws the custom exception
public void ProcessUserInput(string input)
{
    if (string.IsNullOrEmpty(input))
    {
        throw new InvalidUserInputException("User input cannot be null or
empty.");
    }
    // Process the input
}
```

<Bad Code>

```
csharp// Method that throws a generic exception
public void ProcessUserInput(string input)
```

```
{
   if (string.IsNullOrEmpty(input))
   {
      throw new Exception("An error occurred.");
   }
   // Process the input
}
```

In the good example, a custom exception InvalidUserInputException is created to handle a specific error condition, making the code more readable and maintainable. In the bad example, a generic exception is thrown, which provides less context and makes error handling more difficult.

<Memo>
Custom exceptions can be extended to include additional properties and methods, providing even more context and functionality for error handling.

94

Validate method arguments to prevent errors.

Always validate method arguments to ensure they meet the expected criteria before processing.

Validating method arguments helps prevent errors and ensures that your methods operate on valid data, improving the robustness of your code.

< Good Code >

```
csharp// Method that validates its arguments
public void SaveUser(string username, int age)
{
    if (string.IsNullOrEmpty(username))
    {
        throw new ArgumentException("Username cannot be null or empty.", 
nameof(username));
    }
    if (age <= 0)
    {
        throw new ArgumentOutOfRangeException(nameof(age), "Age must 
be a positive number.");
    }
    // Save the user
}
```

<Bad Code>

```
csharp// Method that does not validate its arguments
public void SaveUser(string username, int age)
```

```
{
    // Save the user without any validation
}
```

In the good example, the method SaveUser validates its arguments, ensuring that username is not null or empty and that age is a positive number. This prevents potential errors and ensures the method operates on valid data. In the bad example, no validation is performed, which can lead to unexpected behavior and errors.

<Memo>
Argument validation can be automated using code contracts or validation libraries, which can further simplify and standardize the validation process.

95

Use finally blocks to clean up resources.

Ensure that resources are always cleaned up, even if an exception occurs, by using finally blocks.

Using finally blocks is a best practice for cleaning up resources such as file handles, database connections, or network sockets in your C# programs.

< Good Code >

```
StreamReader reader = null;
try
{
    reader = new StreamReader("example.txt");
    string content = reader.ReadToEnd();
    Console.WriteLine(content);
}
catch (Exception ex)
{
    Console.WriteLine("An error occurred: " + ex.Message);
}
finally
{
    // Ensure the reader is closed, even if an error occurs
    if (reader != null)
    {
        reader.Close();
    }
}
```

<Bad Code>

```
StreamReader reader = new StreamReader("example.txt");
try
{
    string content = reader.ReadToEnd();
    Console.WriteLine(content);
}
catch (Exception ex)
{
    Console.WriteLine("An error occurred: " + ex.Message);
}
// No finally block to ensure the reader is closed
```

In the good example, the finally block ensures that the StreamReader is closed regardless of whether an exception occurs. This prevents potential resource leaks and ensures the application runs efficiently. In the bad example, if an exception is thrown, the StreamReader may not be closed properly, leading to resource leaks.

<Memo>
The finally block is executed after the try and catch blocks, no matter what. It's a feature of the .NET framework to guarantee that cleanup code always runs.

96

Write reusable methods for common tasks.

Create methods for tasks that are repeated throughout your code to improve readability and maintainability.

By writing reusable methods for common tasks, you reduce code duplication and make your code easier to manage and understand.

< Good Code >

```
public class Utility
{
    // Reusable method to format a string
    public static string FormatString(string input)
    {
        return input.Trim().ToUpper();
    }
}
class Program
{
    static void Main()
    {
        string rawString = "  hello world  ";
        string formattedString = Utility.FormatString(rawString);
        Console.WriteLine(formattedString); // Output: HELLO WORLD
    }
}
```

<Bad Code>

```
class Program
{
   static void Main()
   {
      string rawString1 = "  hello world  ";
      string formattedString1 = rawString1.Trim().ToUpper();
      Console.WriteLine(formattedString1); // Output: HELLO WORLD
      string rawString2 = "  csharp programming  ";
      string formattedString2 = rawString2.Trim().ToUpper();
      Console.WriteLine(formattedString2); // Output: CSHARP
PROGRAMMING
   }
}
```

In the good example, the FormatString method in the Utility class centralizes the string formatting logic, making it easy to reuse and update. This improves code maintainability and readability. In the bad example, the string formatting logic is duplicated, making the code harder to maintain and prone to errors if the formatting logic needs to change.

<Memo>
Encapsulating common tasks into reusable methods not only improves code quality but also adheres to the DRY (Don't Repeat Yourself) principle, which is fundamental in software development to reduce redundancy and errors.

97

Use interfaces to define contracts for classes.

Interfaces define a contract that classes must follow, ensuring consistency and promoting loose coupling.

Using interfaces in C# helps to define a set of methods and properties that implementing classes must provide. This promotes a clear structure and allows for more flexible and maintainable code.

< Good Code >

```
csharp// Define an interface with a contract
public interface IAnimal
{
    void Speak();
    void Move();
}
// Implement the interface in a class
public class Dog : IAnimal
{
    public void Speak()
    {
        Console.WriteLine("Bark");
    }
    public void Move()
    {
        Console.WriteLine("Run");
    }
}
// Another class implementing the same interface
public class Bird : IAnimal
```

```
{
    public void Speak()
    {
        Console.WriteLine("Chirp");
    }
    public void Move()
    {
        Console.WriteLine("Fly");
    }
}
```

<Bad Code>

```
csharp// No interface, direct implementation
public class Dog
{
    public void Speak()
    {
        Console.WriteLine("Bark");
    }
    public void Move()
    {
        Console.WriteLine("Run");
    }
}
public class Bird
{
    public void Speak()
    {
        Console.WriteLine("Chirp");
    }
    public void Move()
    {
        Console.WriteLine("Fly");
    }
}
```

In the good example, the IAnimal interface defines a contract that both Dog and Bird classes implement. This ensures that both classes have Speak and Move methods, promoting consistency and making it easier to manage and extend the code. In the bad example, there is no interface, leading to potential inconsistencies and harder maintenance.

<Memo>
Interfaces in C# can also inherit from other interfaces, allowing for the creation of more complex and hierarchical contracts.

98

Leverage inheritance to reuse code.

Inheritance allows classes to inherit methods and properties from a base class, promoting code reuse and reducing redundancy.

Using inheritance in C# enables a class to inherit the functionality of another class. This helps to avoid code duplication and makes the codebase easier to maintain and extend.

< Good Code >

```
csharp// Base class
public class Animal
{
    public void Eat()
    {
        Console.WriteLine("Eating");
    }
    public void Sleep()
    {
        Console.WriteLine("Sleeping");
    }
}
// Derived class inheriting from Animal
public class Dog : Animal
{
    public void Bark()
    {
        Console.WriteLine("Barking");
    }
}
// Another derived class inheriting from Animal
public class Bird : Animal
```

```
{
    public void Fly()
    {
        Console.WriteLine("Flying");
    }
}
```

<Bad Code>

```
csharp// No inheritance, duplicate code
public class Dog
{
    public void Eat()
    {
        Console.WriteLine("Eating");
    }
    public void Sleep()
    {
        Console.WriteLine("Sleeping");
    }
    public void Bark()
    {
        Console.WriteLine("Barking");
    }
}
public class Bird
{
    public void Eat()
    {
        Console.WriteLine("Eating");
    }
    public void Sleep()
    {
        Console.WriteLine("Sleeping");
    }
    public void Fly()
    {
```

```
        Console.WriteLine("Flying");
    }
}
```

In the good example, the Animal base class contains common methods Eat and Sleep, which are inherited by both Dog and Bird classes. This reduces code duplication and makes the code easier to maintain. In the bad example, the Eat and Sleep methods are duplicated in both Dog and Bird classes, leading to redundancy and potential maintenance issues.

<Memo>
C# supports single inheritance, meaning a class can only inherit from one base class. However, a class can implement multiple interfaces, allowing for more flexible design.

99

Use generics to create flexible and reusable code.

Generics allow you to define classes, methods, and interfaces with a placeholder for the type of data they store or use, making your code more flexible and reusable.

Generics in C# enable you to write a class or method that can work with any data type. This reduces code duplication and increases type safety.

< Good Code >

```
csharp// A generic class that can store any type of data
public class GenericRepository<T>
{
    private List<T> items = new List<T>();
    public void Add(T item)
    {
        items.Add(item);
    }
    public T Get(int index)
    {
        return items[index];
    }
}
// Usage
var intRepository = new GenericRepository<int>();
intRepository.Add(1);
int value = intRepository.Get(0);
var stringRepository = new GenericRepository<string>();
stringRepository.Add("Hello");
string text = stringRepository.Get(0);
```

<Bad Code>

```
csharp// A non-generic class that only works with integers
public class IntRepository
{
    private List<int> items = new List<int>();
    public void Add(int item)
    {
        items.Add(item);
    }
    public int Get(int index)
    {
        return items[index];
    }
}
// Usage
var intRepository = new IntRepository();
intRepository.Add(1);
int value = intRepository.Get(0);
// To store strings, you would need a separate class
public class StringRepository
{
    private List<string> items = new List<string>();
    public void Add(string item)
    {
        items.Add(item);
    }
    public string Get(int index)
    {
        return items[index];
    }
}
var stringRepository = new StringRepository();
stringRepository.Add("Hello");
string text = stringRepository.Get(0);
```

The good example uses a generic class GenericRepository<T>, which can store any type of data, making it reusable and flexible. The bad example uses separate classes for each data type, leading to code duplication and less flexibility.

<Memo>
Generics were introduced in C# 2.0 and are a powerful feature for creating type-safe data structures and methods.

100

Encapsulate common functionality in utility classes.

Utility classes group common functions that can be reused across different parts of an application, promoting code reuse and maintainability.

Utility classes in C# are static classes that contain static methods for common operations, reducing code duplication and improving code organization.

< Good Code >

```
csharp// A utility class for common string operations
public static class StringUtils
{
    public static bool IsNullOrEmpty(string value)
    {
        return string.IsNullOrEmpty(value);
    }
    public static string ToTitleCase(string value)
    {
        if (string.IsNullOrEmpty(value))
            return value;
        return
System.Globalization.CultureInfo.CurrentCulture.TextInfo.ToTitleCase(val
ue.ToLower());
    }
}
// Usage
bool isEmpty = StringUtils.IsNullOrEmpty("Hello");
string titleCase = StringUtils.ToTitleCase("hello world");
```

<Bad Code>

```
csharp// Common string operations scattered across the codebase
public class SomeClass
{
    public bool CheckIfNullOrEmpty(string value)
    {
        return string.IsNullOrEmpty(value);
    }
    public string ConvertToTitleCase(string value)
    {
        if (string.IsNullOrEmpty(value))
            return value;
        return
System.Globalization.CultureInfo.CurrentCulture.TextInfo.ToTitleCase(val
ue.ToLower());
    }
}
public class AnotherClass
{
    public bool IsStringEmpty(string value)
    {
        return string.IsNullOrEmpty(value);
    }
    public string ToTitleCaseString(string value)
    {
        if (string.IsNullOrEmpty(value))
            return value;
        return
System.Globalization.CultureInfo.CurrentCulture.TextInfo.ToTitleCase(val
ue.ToLower());
    }
}
```

The good example uses a static utility class StringUtils to encapsulate common string operations, making the code more organized and reusable.

The bad example has these operations scattered across different classes, leading to code duplication and harder maintenance.

<Memo>
Utility classes are often used for operations like string manipulation, file handling, and mathematical calculations, providing a centralized place for common functionality.

101

Use dependency injection to manage dependencies.

Dependency injection (DI) helps manage dependencies in a clean and maintainable way.

Using DI allows you to inject dependencies into a class, making the code more modular and easier to test.

< Good Code >

```
csharp// Good example using Dependency Injection
public interface IService
{
    void Serve();
}
public class Service : IService
{
    public void Serve()
    {
        Console.WriteLine("Service Called");
    }
}
public class Client
{
    private readonly IService _service;
    // Dependency is injected via constructor
    public Client(IService service)
    {
        _service = service;
    }
    public void Start()
```

```
    {
        _service.Serve();
    }
}
// In a DI container setup
var service = new Service();
var client = new Client(service);
client.Start();
```

<Bad Code>

```
csharp// Bad example without Dependency Injection
public class Client
{
    private readonly Service _service;
    public Client()
    {
        // Directly instantiating the dependency
        _service = new Service();
    }
    public void Start()
    {
        _service.Serve();
    }
}
public class Service
{
    public void Serve()
    {
        Console.WriteLine("Service Called");
    }
}
var client = new Client();
client.Start();
```

In the good example, the Client class receives its dependency (IService) through its constructor, making it easier to swap out implementations and test the class. In the bad example, the Client class creates its own instance of Service, making it tightly coupled and harder to test or modify.

<Memo>
Dependency Injection is a core principle of the SOLID design principles, specifically the Dependency Inversion Principle (DIP).

102

Write unit tests to ensure code correctness.

Unit tests help verify that individual parts of your code work as expected.

Writing unit tests ensures that your code behaves correctly and helps catch bugs early in the development process.

< Good Code >

```
csharp// Good example with unit tests
public class Calculator
{
    public int Add(int a, int b)
    {
        return a + b;
    }
}
// Unit test for Calculator class
[TestClass]
public class CalculatorTests
{
    [TestMethod]
    public void Add_TwoNumbers_ReturnsSum()
    {
        // Arrange
        var calculator = new Calculator();
        // Act
        var result = calculator.Add(2, 3);
        // Assert
        Assert.AreEqual(5, result);
    }
```

```
}
```

<Bad Code>

```
csharp// Bad example without unit tests
public class Calculator
{
    public int Add(int a, int b)
    {
        return a + b;
    }
}
// No unit tests provided to verify the functionality
var calculator = new Calculator();
var result = calculator.Add(2, 3);
Console.WriteLine(result); // Manual testing
```

In the good example, the Calculator class has a corresponding unit test that verifies the Add method works correctly. This automated test ensures that any changes to the code can be quickly verified. In the bad example, there are no unit tests, and the functionality is manually tested, which is error-prone and not scalable.

<Memo>

Unit tests are a fundamental part of Test-Driven Development (TDD), where tests are written before the actual code implementation.

Afterword

◆

Thank you for reading through this comprehensive guide on writing cleaner and more readable code in C#.

Throughout these pages, we've explored a variety of techniques, totaling 100, aimed at enhancing the quality of your code.

By leveraging the unique features and idioms of C#, these methods are designed to help you write code that is not only functional but also easy for others to understand and maintain.

Each technique has been illustrated with examples of both good and bad code, providing clear and practical insights into best practices.

Our goal has been to make the content as accessible and useful as possible, ensuring that you can apply these principles directly to your work.

We hope that this book has equipped you with the knowledge and tools to write better C# code and that it will serve as a valuable resource in your ongoing development journey.

Your dedication to improving code readability will undoubtedly lead to more efficient and collaborative coding experiences.

Thank you once again for your commitment to writing better code.